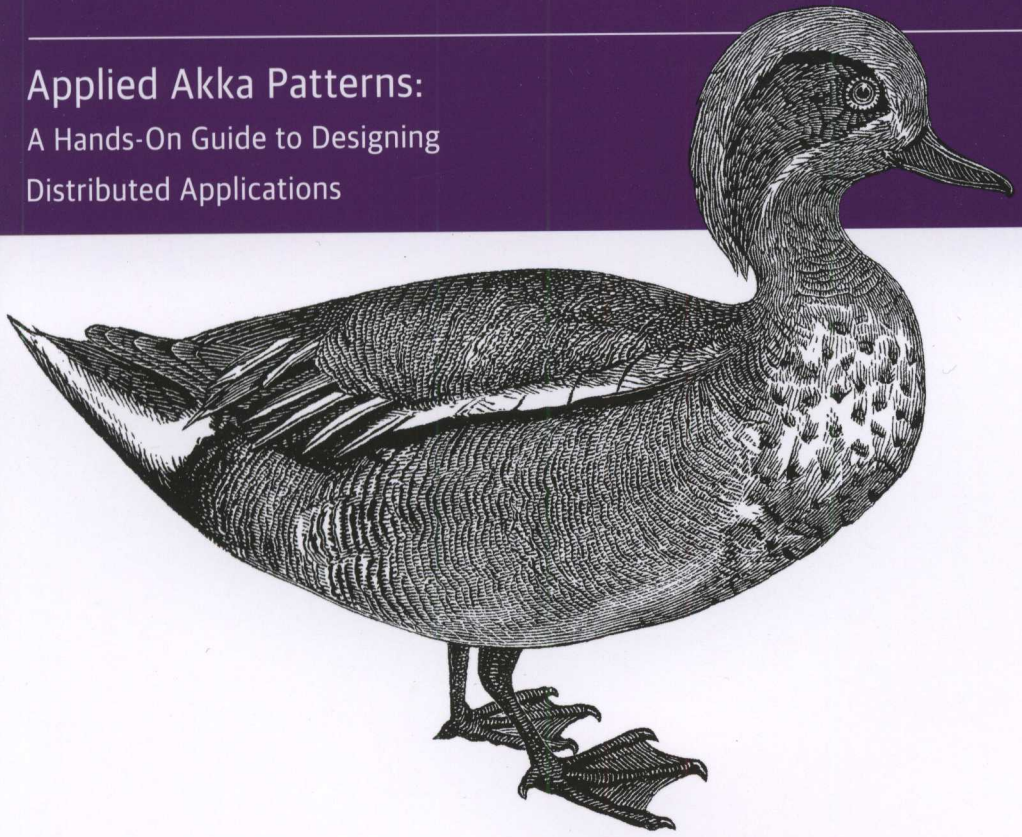
O'REILLY®

Broadview®
www.broadview.com.cn

Akka应用模式：

分布式应用程序设计实践指南

Applied Akka Patterns:
A Hands-On Guide to Designing
Distributed Applications



[美] Michael Nash [加] Wade Waldron 著
虞航仲 译

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Akka应用模式：分布式应用程序设计实践指南

Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications

[美] Michael Nash
[加] Wade Waldron 著
虞航仲 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书一开始介绍了 Actor 模型，因为该模型比其他模型更能反映现实世界。另外，本书介绍了 Actor 模型的一个实现框架 Akka 以及它的工具，而后讨论了在充分利用 actor 架构的基础上使用 Akka 框架来设计软件系统的方法，以及使用它来开发并发性和分布式应用程序的方法。本书还介绍了领域驱动设计（DDD），指导通过结合 Akka、Actor 模型和 DDD 构建强大的、高可扩展的、高可维护的系统。

遵循并灵活运用本书介绍的思路和方法，一定能创建出集可伸缩性、弹性、灵活性、长期易于维护性等优点于一身的应用程序和系统。

版权贸易合同登记号 图字：01-2017-3966

图书在版编目（CIP）数据

Akka 应用模式：分布式应用程序设计实践指南 /（美）迈克尔·纳什（Michael Nash），（加）韦德·沃尔德龙（Wade Waldron）著；虞航仲译 . —北京：电子工业出版社，2017.10

书名原文：Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications

ISBN 978-7-121-32529-8

Ⅰ . ① A… Ⅱ . ①迈… ②韦… ③虞… Ⅲ . ① JAVA 语言－程序设计 Ⅳ . ① TP312.8

中国版本图书馆 CIP 数据核字 (2017) 第 200914 号

策划编辑：孙奇俏

责任编辑：张春雨

封面设计：Karen Montgomery 张健

印　　刷：三河市良远印务有限公司

装　　订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开　　本：787×980 1/16 印张：11.5 字数：255 千字

版　　次：2017 年 10 月第 1 版

印　　次：2017 年 10 月第 1 次印刷

定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”;创建第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了Make杂志,从而成为DIY革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖,共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程,每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人,他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了:‘如果你在路上遇到岔路口,走小路(岔路)。’回顾过去Tim似乎每一次都选择了小路,而且有几次都是一闪即逝的机会,尽管大路也不错。”

——Linux Journal

推荐序

Akka是在Scala语言中实现的一个明星级产品，被广泛地应用在构建基础框架和业务架构中，它把Erlang中被工业界证实为高可用、高容错的Actor模型成功应用到JVM之中，并产生了巨大的影响。我所在的挖财公司，其生产环境中也有一些基于Akka的系统，经过长时间的观察，这些系统运行得都很稳定。

Actor模型本身是一种非常出色的编程模型，但要充分发挥它的优势，还需要开发者对它的各种细节及使用模式非常熟悉。例如，在常见的模式中，有一种“隔离阻塞”模式经常会让初学者很头疼。

actor系统的核心在于各个actor之间的通信，但基于JVM的Akka，其底层仍是基于线程池来工作的，actor之间的通信也是靠线程调度来实现的。因此，当逻辑中含有一些阻塞操作时，需要额外注意并反复确认这些阻塞操作使用的线程与actor通信中使用的线程是否相同，并发请求中多了这些阻塞操作是否会导致所有的线程都被“block”住，进而导致actor之间的通信受到影响。

我自己曾经遇到过一个因为未正确设置隔离阻塞操作而造成严重bug的案例：有一个项目在我自己的笔记本电脑上运行时不会发生任何异常，但当部署到测试环境中运行时却发生了导致整个actor系统不可用的问题。

深入系统调试后发现，系统里有一个作为Router角色的actor，其背后竟然有40个实例在执行同一个会发生阻塞的远程调用（在这种情况下系统至少需要40个可用的线程）。我们知道actor系统底层调度的线程池大小是由并发因子和CPU核心数决定的，我的个人笔记本的CPU核心数是8，测试环境中机器的CPU核心数只有2，两个不同环境的线程池大小相差很大。当设置的并发因子数为10时，系统在我个人笔记本电脑上的线程池大小是80，大于40，因此那40个实例执行远程调用的同时还有空闲的线程可用于actor

之间的通信，系统可以正常运行；而测试环境中的线程数只有 20，是少于 40 的，这使得所有线程都被那 40 个实例的远程调用所占用，系统中没有可用于 actor 之间通信的多余线程，从而导致系统出现了不可用的情况。

除了常见模式，还有一个对于开发者和架构师而言都非常重要的事情——如何集成已有的非 Actor 模型的框架、库或服务。

例如，已有的系统都是基于 Spring 生态的，那么 Akka 应用是否也应该基于 Spring 生态？或者，HTTP 服务究竟该选用运维熟悉的 Tomcat，还是 Akka 自带的 Akka HTTP 呢？这些都是需要技术人员根据实际工程认真思考的问题，不是单单权衡技术利弊就可以做出选择的，可见 Akka 中值得我们去探索的问题还有很多。

这本书对于 Akka 中各个技术点的阐述非常简练，作者在不到 200 页的篇幅中介绍了从 Actor 模型到 Akka 组件，再到领域驱动设计等许多知识点，为读者提供了正确应用 Akka 框架进行分布式设计开发的思路。另外，这本书中有很多干货，内容非常务实，对于应用 Akka 框架进行实际项目开发的人有很大的帮助。

强烈推荐从事 Akka 分布式开发的的技术人员都能仔细阅读这本书，希望有更多的读者能够通过这本书了解 Akka 这个非常强大的工具。

挖财中间件团队负责人，王宏江

2017 年 8 月

译者序

我在大学期间开始接触并使用 Scala，当时就发现了这门语言的强大。正如其名字所表达的那样，Scala 是一门可扩展的编程语言，融合了函数式编程和面向对象编程的特点，支持并发，让异步编程变得很自然，同时表达能力也特别强。虽然 Scala 的学习成本会比较高，尤其和 Golang、Python 这类语言相比，但是这门语言还是非常实用的，对提高开发效率很有帮助。而且 Scala 的创始人 Martin Odersky 说过："Scala 现在是为聪明人创造的，以后也是为聪明人服务的。" Scala 相信程序员的聪明才智，让程序员自行选择合适的结构，以应对千变万化的任务需求，这一点是 Scala 做得很不错的地方。

编写正确的具有容错性和可扩展性的分布式、高并发程序非常困难，主要是因为我们使用了错误的工具和错误的抽象层次，而 Akka 的出现改变了这种情况。Akka 是参照 Apache 2 许可证（一种公认的开源许可证）发布的开源项目，通过使用 Actor 模型来提高抽象的级别，并提供了一个更好的平台来构建可扩展的、有弹性的、响应式的应用程序，详细信息可参阅 *The Reactive Manifesto*。对于容错，Akka 采用"让它崩溃（Let It Crash）"的模式，这种模式可以帮助构建可自我修复和永不停止的软件系统。其中 actor 还提供了透明的分布式抽象，以及真正的可扩展与高容错应用的基础设施。

毕业后，我在腾讯微信团队的后台架构部从事软件开发工作，作为腾讯的战略级产品，微信平均每天面临亿级的在线用户。面对这种用户规模的挑战，基本每一行代码都需要考虑高并发和分布式的场景，大道至简的思想基本贯彻在整个技术产品线上。例如，大系统小做，让一切可扩展；剥离复杂，让剩下的更简单；在容灾之前面向最坏情况思考，防止雪崩；精细监控，迅速响应等。微信团队内部的很多技术点也和 Actor 模型很像，比如微信最核心的消息模型和 Actor 的邮件模型就很有渊源。消息被发出后，会先在后台临时存储，为了使接收者能更快接收到消息，系统会推送通知给接收者，最后客户端主动到服务器端拉取消息。当然整个微信架构就是微服务的架构，每一个请求后面可能

会涉及几百个服务。如何扩展、如何高容错、如何弹性，这些基本是每天都会遇到的挑战，并且也都是设计和开发系统的时候需要考虑的事情。

后来，我离开腾讯，在猎豹移动的广告系统架构部以及新闻团队从事系统架构开发工作，同样涉及微服务架构，每天都要考虑到这些分布式系统可能遇到的并且需要处理的问题。尤其广告系统涉及金钱，因而需要严格保证其高可用性和一致性。

现在，我在创业公司担任技术负责人，同时负责公司内部多个系统的架构工作，也需要时时刻刻考虑和处理高并发、高容错等分布式问题，同样用到了高并发的分布式微服务架构。

本书原名是 *Applied Akka Patterns : A Hands-On Guide to Designing Distributed Applications*，书中介绍了一些很好的分布式系统的设计原则，而且也介绍了 Actor 模型和 Akka 工具包，对于使用 JVM 体系结构的开发者来说是非常值得一看的，因为通过本书能很快地学习并掌握一个强大的工具，在工作中提高生产效率。对于那些使用非 JVM 体系结构的开发者来说，通过这本书能了解到一个更强大的工具，也是极有帮助的。同时，本书介绍的很多指导原则对于分布式系统设计有很大的借鉴意义，可以避免让自己陷入困境。这本书是一个起点，帮助我们发现新大陆，但绝不会是我们的终点，开拓这块新大陆还需要自己不断努力。其实 Scala 本身就是一门很强大的语言，最近也因为 Spark、Kafka 等项目在国内掀起了一波关注热潮。Akka 也很优秀，以至于被 Lightbend 收购，并直接用 Akka 的 actor 替代了 Scala 本身的 actor。总之，本书中介绍的内容都是非常值得探索和学习的。

决定翻译这本书，不仅因为我参与和主导了不少分布式的项目，也因为对分布式系统设计和开发的热爱，以及对 Scala 语言本身的喜欢。我抱着把 Actor 模型以及 Akka 传递给中国的工程师并让更多人能接触、了解它们的态度来尝试翻译这本书。虽然它们在国内还不算太火，但是在国外已经非常受欢迎。

这本书虽然只有 160 多页，但是翻译过程还是比较辛苦的，很多地方的意思都比较隐晦，用中文直译会很有挑战。所以我平时会留意大家在社区里讨论的内容，参考社区里的资料，在遇到一些和自己项目经验不太一致或者比较不确定的地方，也会尝试去查看 Akka 的源码来尽量保证自己的理解无误，避免误导读者。

在本书的翻译工作结束之际，我首先要感谢博云科技的 CTO 李亚琼老师，他的引荐让我获得了翻译这本书的机会。还要感谢孙国立、曾杰瑜、付冉、刘岸，他们在百忙中帮我审阅了本书翻译稿的大部分章节，并针对涉及的专业概念提出不少修改意见。最后要感谢本书的策划编辑孙奇俏，她为本书的编辑和校对做了大量细致的工作。

翻译过程中虽力求理解作者意图，把握全文，但是难免会发生错误，若广大读者发现错误，我在此深表歉意。欢迎广大读者及时与我和出版社联系，提交勘误，方便后续读者更好地阅读。如果有任何好的想法和建议，也欢迎和我邮件沟通，我的邮箱地址是hangzhong.yu@gmail.com。

虞航仲

2017 年 8 月于北京

目录

前言

响应式应用开发是软件开发的新前沿。 随着联网设备的普及，数据量也在增加。以前的单线程批处理数据的旧技术根本无法满足这个新领域所提出的需求。大数据的概念已经兴起，我们需要新的工具和新的技术来应对它。

通常，解决现有问题的灵感并不是来自现有的技术，而是来自过去的经验。许多现今用于处理大数据的新工具实际上是基于旧的 actor 的概念而产生的。actor 是构建 Akka 的关键概念，但其根源追溯起来应该属于过去。actor 不是一个新概念，相反，它是一个被重新关注的旧概念。

当开始探索 Akka、actor、streams 和其他与之相关的技术时，我们将从现实世界的角度来看待它们：如何在一系列项目中安排一组人，同时优化他们的可用时间及技能？这是一个复杂的问题，并不是只用一个下午就可以解决的。但这又是一个有趣的问题，为深度探索提供了很大的空间。这也是大多数软件开发人员在职业生涯中的某个时刻一定会遇到的问题。 在对 Akka 进行探索的过程中，我们将回顾这个问题。

在解决问题之前，我们必须先了解可用的工具，还需要了解这些工具为什么存在，以及它们可以解决什么样的问题。我们需要知道 Akka 的起源及其在 Actor 模型中的根源。我们需要一套指导原则将应用程序拼接在一起，这套原则会在探索域驱动设计（DDD）的过程中被发现。有了这些基础，便可以开始使用 Akka 提供的所有工具来构建域了。我们可以探索简单的 actor 的使用方法以及它如何与流关联，可以让系统分布在多个节点上，使其具有更好的容错性、可用性及可扩展性。

首先，我们需要知道这一切的根源在哪里。

本书使用的排版约定

本书使用如下排版约定：

斜体（*Italic*）

标志着新词汇、URL、邮箱地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序清单（包括段落中的），表示程序元素，如变量名或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽字体加粗（**`Constant width bold`**）

显示字面上被用户输入的命令或其他文本。

等宽字体且斜体（*`Constant width italic`*）

显示应该被用户提供的值或者由上下文决定的值所替换的文本。

该图标表示技巧或建议。

该图标表示一般注释。

该图标表示警告或者注意事项。

O'Reilly Safari

Safari（以前的 Safari Books Online）是企业、政府、教育者和个人的会员制培训及参考平台。

订阅者可以从一个完全可搜索的数据库中获得来自 250 多家出版商提供的成千上万的书籍、培训视频、互动教程，这些出版商包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley&Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones&Bartlett、Course Technology。

若想获得更多资讯，请访问 *http://oreilly.com/safari*。

如何联系我们

请将对本书的评价和发现的问题通过如下地址通知出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们提供了本书网页，上面列出了勘误表、示例和其他信息。请通过 *http://bit.ly/applied-akka-patterns* 访问该页。

要给出本书意见或者询问技术问题，请发送邮件到 *bookquestions@oreilly.com*。

更多有关书籍、课程、会议和新闻的信息，请见网站 *http://www.oreilly.com*。

在 Facebook 找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看：*http://www.youtube.com/oreillymedia*

致谢

这本书能够出版，要感谢许多人的帮助，包括 Lightbend 和社区里的 Akka 团队，本书的编辑 Nan Barber 以及评论家 Konrad Malawski、Sean Glover、Petro Verkhogliad。特别要感谢我们的家人给予的宽容和支持。

读者服务

轻松注册成为博文视点社区用户（*www.broadview.com.cn*），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 *提交勘误* 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 *读者评论* 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：*http://www.broadview.com.cn/32529*

第1章

Actor模型

理解如何正确使用 actor 是最基本的，这样才能使其发挥最大的作用，这也是我们将在本章中学到的知识。本章将会探讨 actor——它是如何工作的，它们如何与彼此及外部世界进行交互。

设计软件时用到的很多技术都教导我们，在编写代码之前先去了解现实世界是很有必要的。我们必须了解将要开发的软件用例，知道谁将使用它，如何使用它，这些信息对设计优秀的软件系统而言至关重要。还有一个非常重要的问题在设计过程中经常被忽略：系统执行需要耗费多长时间？任何一个开发过高并发系统的工程师都知道，时间是整个软件开发过程中的重要组成部分。

下面先探讨一些看似与软件开发并不密切相关的话题，不用担心，稍后会向大家说明这些是如何与软件开发产生联系的！

现实是最终一致的

思考一下平时去拿一杯咖啡的过程，表面上看，这是一个很简单的过程：伸手，然后拿到一杯咖啡。这个过程并没有包含太多动作，但是，更深入地去思考这个过程会发现许多奥秘。

为了能拿到这杯咖啡，首先需要知道它在哪里。环顾四周然后会看到它，但是此时看到的这杯咖啡是它现在的状态吗？可能已经是它曾经的状态了？因为我们是基于这个杯子反射回来的光来判断它的位置的，但是光是需要时间传播的。而且，当眼睛接收到这些数据之后，也需要时间先来处理数据然后再传送到大脑。另外，神经系统的其他部分还需要进行额外的一系列处理，才能让我们移动手臂，最后拿起这杯咖啡。

这个过程中的每个阶段都会产生一些时间延迟，而这些延迟最终都会影响到拿杯子这件

事情本身。如果处于一个静止的环境中，这点延迟对于简单的拿杯子的动作而言不会有太大的影响，然而世界本质上是动态的，时刻都在发生变化，随着事件频繁发生而剧烈变化，这些小的延迟便会积累。不过对于前面那个简单的拿起一杯咖啡的例子来说，这点延迟还是相当微不足道的。那么，如果想要在杯子从桌子上掉下来的时候尝试去抓住它，同时又保证咖啡不洒出来，是不是就变得相当有挑战性了呢？如果想要同时抓住很多正在掉落的杯子呢？我们现在仿佛处在一个不可能完成任务的境界之中。

事实上，现实世界是受制于光速的。物理定律和光速为因果关系规定了上限。只有当两件事情占据相同的空间（当然这是不可能的），并且它们发生的时间有一定间隔时，先发生的事件才能对后发生的事件产生影响。当两个人在不同的距离点观察同一件事件时，其实他们会在不同的时间点经历这件事，距离近的人会稍早于距离远的人。然而，尽管经历的时间点不同，但都是真实地经历了。Actor 模型就是基于这种现实物理世界的规律而设计的。

我们生活在处理一些过时信息的状态中。比如，细胞之间通过激素进行消息交流。再比如，我们日常聊天、看新闻、阅读博客，所有这些不同信息间的交流都是以异步的形式进行的。事实上，最后会发现生活中并没有事情是同步进行的。

即便是计算机，其行为也是异步的，计算机的每一步操作都是通过在某种介质上传播信号来完成的，信号可以是电信号、光信号或者其他信号。

既然世界上所有的事情都是以某种异步的形式在进行着，那么为什么还要花大力气去尝试编写同步的软件系统呢？我们也经常告诫自己，软件系统应该基于现实世界来建模和构造，但却忽略了时间这一基本概念。如果基于现实世界来建模，同时使用异步事件或消息机制来构造软件系统，岂不是更好吗？

反过来思考一下，如果通过传统的同步软件系统来模拟现实世界的话，这个世界会变成什么样子？最后的结果又会是什么？

回到拿一杯咖啡的例子。当大脑决定要去拿一杯咖啡的时候，首先需要暂停时间，至少是暂停我们所处环境的时间。停止周围的世界，这样才能保证在决定要拿一杯咖啡的那一刻到喝到咖啡的那一刻之间没有发生任何变化，没有任何事情影响到这个过程。如果快要拿到咖啡的时候，突然出现其他人抢先把这杯咖啡拿走了，那便会造成很混乱的局面。所以，只有冻结周边的一切，把状态彻底锁定，才能保证不会发生类似的混乱。任何想要拿这杯咖啡的人都会被“冻结”，直到我们成功拿到这杯咖啡为止。当然这并不意味只是冻结另外一个想拿这杯咖啡的人的状态，还包括空气、阳光以及这个杯子周边的一切。我们和杯子之间的所有影响因素都需要被冻结，否则就可能拿不到这杯咖啡，或者最后拿到的咖啡可能已经不是一开始看到的咖啡了。这听起来很复杂，比基于时间

建模的系统要复杂得多。

这是 Actor 模型的基础之一——考虑时间因素，我们才能编写可以反映现实世界实际状态的软件系统，而不是去假设一个并不存在的时间被冻结的世界。

解构 Actor 模型

1973 年，Carl Hewitt、Peter Bishop 和 Richard Steiger 一起开始帮忙解决论文 *Universal Modular Actor Formalism for Artificial Intelligence* 中提到的一些问题。虽然许多编程范例都是基于数学模型的，不过正如 Hewitt 所说，Actor 模型的灵感是来自物理学的，它经历了多年的演变，但是基本概念却一直保持不变。

重要的是，在使用 Akka 时，可以在不使用 Actor 模型的情况下编写代码。而且事实上使用 actor 并不等同于使用 Actor 模型。有很多开发者在使用 Akka 多年后仍没有意识到 Actor 模型的存在。

使用 Actor 模型和简单使用 actor 之间的区别在于对待 actor 的方式。如果把 actor 当作顶级构建模块使用，所有系统程序代码都在 actor 系统内编写，这种情况下就是在使用 Actor 模型。另一方面，如果构建的系统里面有在非 actor 模块中的 actor，即部分程序代码不是在 actor 系统内编写的，那么便不是在使用 Actor 模型。

还有一点也很重要，Actor 模型中的编程并不是一种工具或技术。整个编程语言都是围绕 Actor 模型的思想实现的，所以它更像是一种编程范式而不是工具集合。学习 Actor 模型编程就和学习面向对象编程或函数式编程一样，由于 Actor 模型是早于 Akka 的，所以除了 Akka，还有很多其他版本的 Actor 模型的实现。

例如，Pony 语言就是基于 actor 并且可以很容易地实现 Actor 模型的一种编程语言。另外，Erlang 进程也相当于 Akka 的 actor，它们在 Erlang 里面是非常基础的功能特征。再比如 Ada 编程语言，因为有 Task 的概念和名为“entries”的消息存在——消息由异步机制的 Tasks 控制来实现排队——这意味着 Ada 也是可以用来实现 Actor 模型的。

实现 Actor 模型需要遵循以下几个基本规则。

- 所有的计算都是在 actor 中执行的 。
- actor 之间只能通过消息进行通信交流。
- 为了响应消息，actor 可以进行下列操作。
  - 更改状态或行为。
  - 发消息给其他 actor。
  - 创建有限数量的子 actor。

如果熟悉 Akka，可以马上知道这些是如何在 Akka 里面实现的。但我们要先抛开 Akka，直接探讨 actor，这样才能更好地理解最基本的 Actor 模型和基于 Akka 实现的 Actor 模型之间的差异。

所有的计算都在一个 actor 中执行

Carl Hewitt 称 actor 为基本的计算单位[1]。基本的计算单位是什么意思呢？这意味着使用 Actor 模型构建系统时，一切都是 actor。无论是计算斐波纳契序列还是维护系统中用户的状态，都可以在一个或多个 actor 中进行。

实现“一切都是 actor”的想法并不是一件简单的事情。如果每次计算都需要在一个 actor 中进行，这意味着每个函数和每个状态变量都可以是 actor。即使在技术上可行，也并不实用。通常，如果有一组相关的函数，我们会将这些函数都封装到一个 actor 中。这样做并不违反 Actor 模型，但是怎么划定那条用来判断是否违反 Actor 模型的边界线呢？后面学习领域驱动设计（DDD）时，将更详细地讨论这一点。在介绍 DDD 时引入的构建模块是非常适合转化成 actor 的，可以将其作为创建 actor 的指导原则。

Actor 模型中的 actor 不仅有状态，还有行为，这听起来很像 OOP 的定义。其实两者是密切相关的，Alan Kay 创造了“面向对象编程”这个术语，同时他也是 Smalltalk 语言的原始创造者之一，这很大程度上受到了 Actor 模型的影响。虽然 Smalltalk 和 OOP 的最终演化远离了 Actor 模型，但 Actor 模型的许多基本原则仍然在影响着现在的 Smalltalk 和 OOP。事实上，OOP 最初的核心并不是对象本身而是对象之间的消息流。

OOP 将对象（类的实例）作为基本的计算单位，Java 或类似语言的开发者会比较熟悉。另一方面，函数式编程则是围绕函数及其应用程序来进行计算机运算的，这一点在 Lisp 和 Haskell 语言中可以体现出来。

Actor 模型中另一个对高并发应用有帮助的是隔离 actor 状态的思想。actor 的状态永远不会直接暴露在外面，也无法被其他 actor 查看或修改，除非通过消息机制间接地进行。这种隔离机制同样适用于 actor 的行为。actor 内部的方法和机制同样也不会直接暴露给其他 actor。事实上，在 actor 内部，状态和行为可以被视为相同的因素（后面会更详细地介绍）。

模型中的 actor 可以有许多不同的形式。它们可以作为高技术性的结构存在，如数据库

1 更多有关信息请参阅视频“Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know...)”（*http://www.youtube .com/watel?v=7erJIDV_Tlo*）。

访问层；也可以作为与特定领域相关联的结构存在，如一个人或一个日程表，甚至可以进行简单的数学运算。系统中需要执行任意计算的任意结构都可以是一个 actor。

大家将会在后面的章节中发现，actor 是 Actor 模型以及基于 Akka 的应用程序中最基本的构建块。

actor 之间只能通过消息进行通信

我们以互相隔离的方式创造了 actor，使它们永远不会暴露自己的状态或行为。这是 Actor 模型很重要的一个特性。但是因为用这种方式隔离了它们，因此需要去寻找其他可以与它们进行通信并了解系统状态的方法。

在 Actor 模型里面，所有的通信都是基于消息机制进行的，这也是 actor 之间进行通信的方法。

每个 actor 在创建时都会获得一个地址，该地址是与该 actor 进行通信的入口。不能通过这个地址直接访问该 actor，但可以通过这个地址发消息给它。

Akka 区分了地址和引用的概念。大多数 actor 通信都是通过引用完成的。Akka actor 也有地址可以在某些情况下使用，然而通常会避免这样使用。无论使用引用还是地址，基本原理都是一样的：通过一种方法找到 actor 的邮箱，以便向其发送消息。

发送给 actor 的消息是不可变的数据。这些消息会被发送到目标 actor 提供的地址并被存在邮箱里面。消息到达邮箱后的状态在发送方的控制范围之外。可以不按发送顺序成功投递消息，但投递也有可能失败。Actor 模型提供了最多投递一次的消息投递机制，这意味着有可能发生投递失败的情况。如果想要确保每次投递都能成功，则需要使用其他工具来辅助。

最多投递一次是 Actor 模型和 Akka 提供的默认交付保证。然而，我们可以基于最多投递一次的机制来构造最少投递一次的机制，Akka 中的某些机制可以实现这一点。

Akka 进一步提供了更强大的顺序保证机制，可以确保正在通信的 actor 之间的消息都是按顺序被送达的。也就是说，一个 actor 发给另外一个 actor 多个消息的时候，可以确保消息被送达的顺序和发送的顺序是一样的。

消息被投递到邮箱后，actor 可以接收并处理这些消息，但是每次只能接收处理一条消息。

没有一个 actor 可以同时接收处理两条或两条以上的消息。这一点很关键，确保了我们可以以单线程的模式在 actor 中进行操作。actor 可以很自由地修改自己的内部状态，而不用担心是否会有其他线程也在操作该状态。只要维持这样的单线程模式，状态就可以避免遇到并发问题。

消息的确切类型取决于 actor 的功能。对于不同领域的 actor，经常会看到消息使用与领域相关的专业术语当作命令或事件，比如项目调度的例子里面会出现如“AddUser”或者“CreateProject”这样的命令，如果是一个更具技术感的 actor，相关的消息也会更有技术范儿，如“Save”和“Compute”。

因为 actor 之间只通过消息机制进行通信，因此产生了一些有趣的可能。只要邮箱另一端的 actor 能处理这条消息，那么便可以用任何方法完成这个工作 (见图 1-1)。这意味着接收端的 actor 可以直接处理这条消息，或者把自己当作这条消息的中间代理，只简单地把该消息转发给另外一个 actor 去进行必要的处理即可。接收端 actor 也可以把该消息切分成很多更小的消息块，然后发送给其他 actor 进行进一步的处理。以这种方式，消息的处理细节可以根据需要而简单化或复杂化，并且这些细节对于消息的发送者而言是透明的。

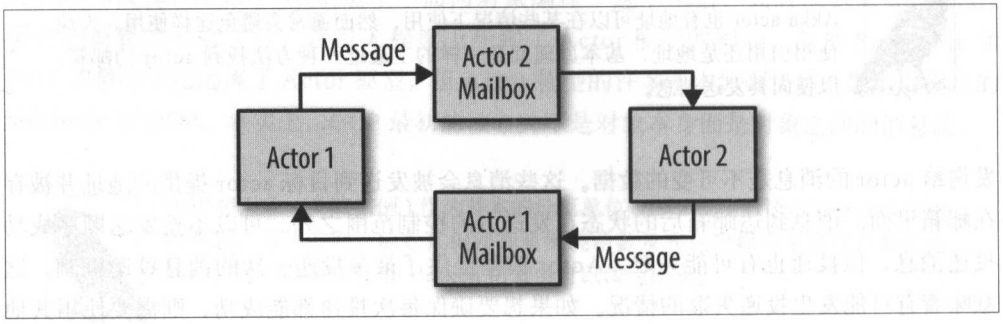


图1-1　actor之间互相通信

值得一提的是，Akka 的邮箱略有不同。Akka 默认提供可以保证顺序的邮箱机制，这样可以保证消息按被投递的顺序接受处理。

actor 可以创建子 actor

在 Actor 模型中，一切都是 actor，而且 actor 之间只能通过消息机制进行通信，但是

actor 还得知道其他 actor 的存在。

当 actor 接收到消息后，可以进行的操作之一是创建有限数量的子 actor。之后父节点就会知道它的所有子节点的存在，并可以访问子节点的地址。这意味着父节点可以向子节点发送消息。

除了通过创建子节点来获知其他 actor 的方法，一个 actor 还可以把地址信息通过消息机制发送给其他 actor。这样父节点便可以把它知道的所有 actor（包括自身）的地址信息都通知给子 actor，所以子 actor 可以很容易获取到父节点或兄弟节点的地址。只需要进行一点简单工作，子节点就可以知道它所处的层级体系里有哪些其他 actor。此外，如果用于 actor 的地址都遵循一个固定规则，那么其他 actor 的地址也可以按照该规则去计算合成，但是如果不小心使用，可能会造成不必要的复杂性问题以及安全隐患。

actor 的这种层级结构意味着，除根节点以外的所有节点都将拥有一个父节点，同时任何节点都可以拥有一个或多个子节点。这样从根节点开始遍历整棵树的 actor 集合被称为一个 actor 系统。

actor 系统中的每个 actor 总是可以通过其地址被唯一标识。其实地址的命名并不需要遵守任何特定的模式，只要地址是唯一的，可以用来唯一标识一个 actor 即可。Akka 使用层级结构的模式来命名，就像目录结构一样（如图 1-2 所示），或者也可以使用随机生成的唯一键。

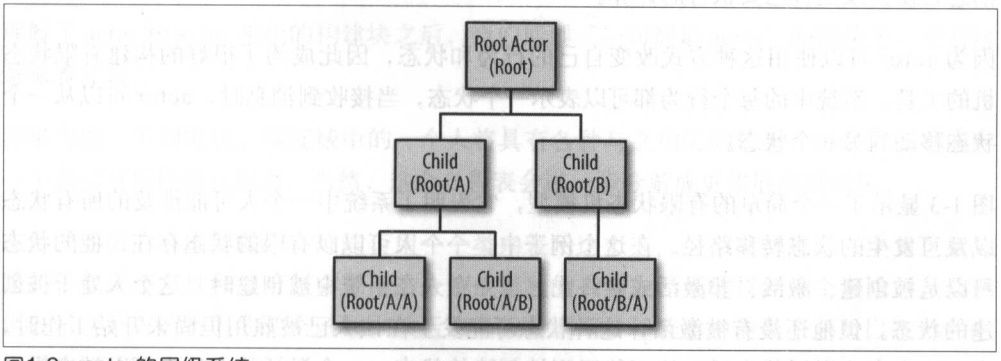


图1-2　actor的层级系统

如图 1-2 所示，根 actor 是顶级的 actor，其他的节点都在它的下面被创建。根节点下有两个子节点 A 和 B，A 的地址是 Root/A，B 的地址是 Root/B。A 下面也有两个子节点，同样分别被命名为 A 和 B。即使它们共享了其他 actor 的名字，但地址仍是唯一的（Root/A/A 和 Root/A/B）。子节点 B 也有一个地址为 Root/B/A 的子节点，同样，地址是唯一的，即使名称不唯一。以上是一个介绍如何生成 actor 地址的例子。

在 Actor 模型里，子 actor 是最有用的集成技术之一，在后续的章节中也会看到，这也是 Akka 中 actor 监督机制的基础。

actor 可以改变自己的状态或行为

除了发送消息和创建子 actor，actor 还可以改变它们对下一个消息的反应。当谈论一个 actor 如何改变时，我们经常会使用“行为”这个术语，其实这很容易误导人，因为行为的变化或 actor 反应的变化也可以表现为状态的变化。

来看看前文中讨论的调度示例。假设有一个代表个人可用性的 actor，初始状态可能表明这个人可完成一个项目。当一条指派这个人到某个项目的消息送达后，actor 会改变自己的状态，这样当下一条消息到达后，它会显示该人的状态是不可用的。即使这是状态的变化，我们仍然认为这是 actor 行为的改变，因为当下一条消息到达时，该 actor 已经表现为不可用了。

我们还可以让 actor 在接收到下一条消息时改变自己将要执行的计算，表示该系统中的人的 actor 可以以不同的状态存在着。当一个人受雇并可以被分配到某个项目中时，他们会以激活的状态存在。但是，某些条件可能导致这个人转移到非激活的状态（终止雇佣合同、延长休假等）下。处于激活状态下，这个人可以正常处理项目请求。但是，处于非激活状态下，系统可能会拒绝项目请求。在这种情况下，该 actor 在接收到下一条消息时便会改变自己要执行的计算。

因为 actor 可以使用这种方式改变自己的行为和状态，因此成为了很好的构建有限状态机的工具。系统中的每个行为都可以表示一个状态，当接收到消息时，actor 可以从一个状态移动到另一个状态。

图 1-3 显示了一个简单的有限状态机模型，它表明了系统中一个人可能涉及的所有状态以及可发生的状态转移路径。在这个例子中，一个人可以以有限的状态存在，他的状态可以是被创建、激活、非激活或者终止。当一个人在系统中被创建时，这个人处于被创建的状态，但他还没有被激活，这种状态可能发生在该人已被雇用但尚未开始工作时。当一个人变成激活状态时，他不能回到被创建的状态。一个激活状态的人可以被安排到某个项目中执行工作。在某个时间点，一个人可能会变成非激活的状态，这也许是因为他将要休假了。在非激活状态下，该人不能接收处理任何请求，但可以在激活和非激活状态之间自由转移。最终，该人可能离开这个公司，这时系统会把他的状态变成终止状态。在一个人进入终止状态后，他不能回到激活或非激活状态，但可以从终止状态转移到被创建的状态，因为他可能又被这家公司雇用了。

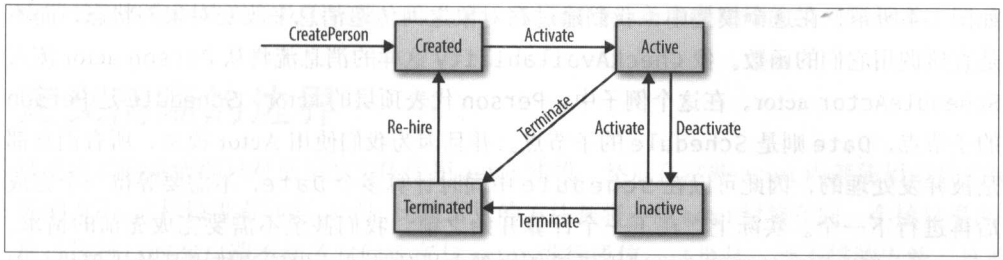


图1-3 基于actor的有限状态机

actor 是一种用来构建有限状态机的很好的方式。Akka 提供了特定的工具，通过 Akka FSM 的形式更容易创建有限状态机。

我们显然可以创建更复杂的场景，但这些已经足够让大家知道通过改变 actor 的行为可以实现的事情的类型。无论是改变接收到的消息，改变处理消息的方式，还是改变 actor 的状态，这些都属于改变行为的范畴。后面将详细讨论 Akka 提供的几种成熟的用于改变 actor 行为的方法。

一切都是 actor

理解了 actor 和 actor 系统的构建块之后，我们回到“一切都是 actor”的想法上，看看这意味着什么。

再来考虑一下调度域。调度域中的一个人将具有各种与之相关的信息，每个人都可能有一个表示可用性的日程表，当然，这个日程表会进一步分解成更离散的时间段。

在传统的面向对象架构中，可能会用一个类来表示这个人。这个类将具有一个与之相关联的日程表类，日程表可能会进一步分解成单独的日期。当系统收到一个请求后，会调用表示这个人的类中的某个函数，然后该函数会调用日程表类中的某个函数以及单独的日期数据。

除用一个 actor 来表示人之外，Actor 模型与上述情况并没有太大的不同。其实在 Actor 模型里面，日程表也可以是一个 actor，因为它有可能会进行一些计算。每个单独的日期也可以是一个 actor，因为它们也可能需要参与计算。实际上，请求本身也可能有一个与之相关联的 actor，因为在某些情况下，请求本身可能需要聚合由模型的其他部分计算出的信息。在这种情况下，我们需要创建一个处理聚合操作的 `RequestWorker`。

如图 1-4 所示，在这个模型中，我们通过在对象之间传递消息来改变对象的状态，而不是直接调用它们的函数。像 `CheckAvailability` 这样的消息流将从 `Person` actor 流入 `ScheduleActor` actor。在这个例子中，`Person` 代表顶层的 actor，`Schedule` 是 `Person` 的子节点，`Date` 则是 `Schedule` 的子节点。并且因为我们使用 Actor 模型，所有消息都是被并发处理的，因此可以在 `Schedule` 中同时计算多个 `Date`，不需要等待一个完成后再进行下一个。实际上，在下一个计算开始之前，我们甚至不需要完成先前的请求。`Schedule` 可以同时处理两个尝试查看重叠日程安排的请求，但因为模式是单线程的，因此当两个请求尝试修改同一天的日程安排时，只有第一个请求会成功。这是因为用同一个 actor 处理该日期的两个请求时，该 actor 每次只能处理一个请求的消息。另一方面，如果请求修改的日期没有重叠，这两个修改请求可以同时被处理和完成，这使得我们能够更好地利用资源。

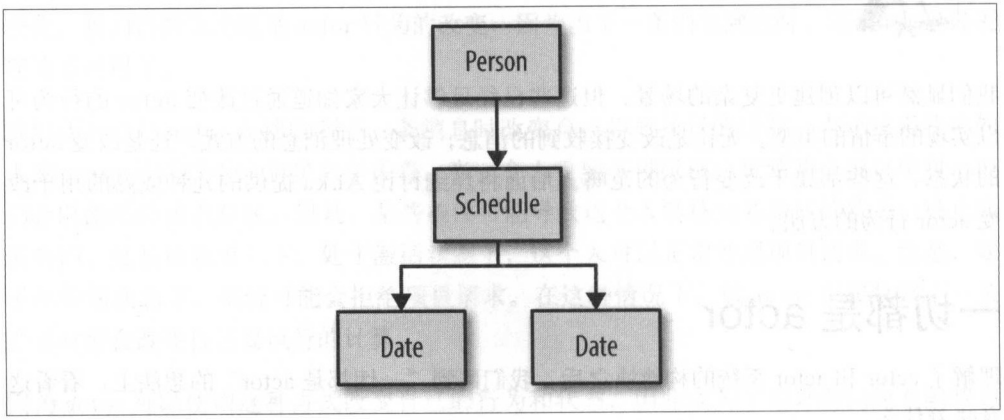


图1-4　基于actor的人员调度系统示意图

Actor 模型的使用

Actor 模型是一个强大的工具，使用恰当时可以帮助我们构建高度可扩展、高度并发的应用程序。但是与任何其他工具一样，它并不是完美的。如果使用不当，Actor 模型可能会创建难以追踪的复杂代码，甚至会变得更难调试。学习完这些工具后，你可能会决定在应用程序的所有地方都使用 Actor 模型，从而创建一个真正的 Actor 系统。但是早期还是先简单尝试然后让自己习惯它比较好。后面的章节将进一步探讨这些想法，并提供关于 actor 构建的其他指导。

有一件关于 Actor 模型的事需要大家知道：和其他技术一样，有时我们可能想要应用它，但有时可能不想。在某些情况下，选择其他模型可能更适合手头的任务。在这些情况下，不要害怕去选择使用其他工具或技术。Actor 模型并不是或全有或全无的命题，关键是

弄清楚需要它的地方以及不需要它的地方，然后创建一条清晰明确的线分隔两者。

定义清晰的边界

许多成功的系统都只对顶层的实体使用 actor 建模，然后在这些 actor 内部使用函数式编程的方法，这样做没有问题。此外，比较常见的方法是将一组 actor 封装在同一个接口里面，使得该接口的客户端不知道它们正在与 actor 进行通信，这也是一种不错的方法。这些方法都实现了创建清晰的界限。第一个例子里面只在上层拥有 actor，actor 里面所有的实现都是使用函数式编程完成的；第二个例子里面，接口本身就是 actor 和非 actor 之间的清晰界限。

如果决定在应用程序的部分功能内使用 Actor 模型，那么应该在该部分功能的所有范围内都坚持使用，而只在明确的边界上脱离它。这些边界可以是前端和后端之间的边界，也可以是应用程序中两个不同的微服务之间的边界。有很多方法可以分割应用程序，也有很多方法可以应用 Actor 模型。但是应该避免破坏 Actor 模型本身所在之处的上下文，因为在不清楚的模式或理由下切换代码风格很容易导致代码杂乱和复杂。我们一定不想在同一个上下文中频繁跳进跳出 Actor 模型，相反地，会希望寻找从系统边界或业务边界跳进或跳出 Actor 模型的方式。

来看一个具体的例子。在调度域中，我们可能需要构建一个库来处理调度任务。库里面更倾向于使用 Actor 模型，但在外面打算使用其他技术。这是系统边界的一个很好的例子，可以从一个计算模型转换到另一个模型。

处理这种情况的一种方法是将所有任务都当成 actor 暴露出来，这样外部的客户端代码就会充分意识到它们对接的是 actor。在这种情况下，会得到一个如下所示的消息协议。

```
object ProjectProtocol {
  case class ScheduleProject(project: Project)
  case class ProjectScheduled(project: Project)
}
```

然后客户端会使用如下代码来发送消息。

```
def scheduleProject(details: ProjectDetails): Future[Project] = {

  val project = createProject(details)

  val result = (projects ? ProjectProtocol.ScheduleProject(project))
    .mapTo[ProjectProtocol.ProjectScheduled]
    .map(_.project)
```

```
    result
  }
```

这样是可以的，但还不够完善。问题在于，actor 代码里混杂着其他代码，这些代码可能具有不同的并发机制，从而引发一些意想不到的问题。这个过程实际上包括以下几个步骤：首先创建一个项目，然后安排项目，最后执行其他操作。这几个步骤都会涉及调度系统，即都和 actor 有联系。然而，代码中可能还有其他部分与不使用 actor 的上下文进行通信，所以在这里也可以存在函数调用或者 future 等机制。这种不断跳进和跳出 Actor 模型的行为若不能很好地被隔离，系统会变得很难跟踪。在这种情况下，更好的方法是创建一个 API 来封装 actor，代码如下。

```
class ProjectApi(projects: ActorRef) {
  def scheduleProject(project: Project): Future[Project] = {
    val result = (projects ? ProjectProtocol.ScheduleProject(project))
      .mapTo[ProjectProtocol.ProjectScheduled]
      .map(_.project)
  }
}
```

这层包裹在 actor 外的封装相当于一层与外界隔离的绝缘层，是用来隔离 actor 和非 actor 的明确边界。此时若要使用此 API，代码如下。

```
def scheduleProject(details: ProjectDetails): Future[Project] = {

  val project = createProject(details)

  val result = projectApi.scheduleProject(project)

  result
}
```

代码变得更短了，我们成功地把复杂的程序移到了一个单独的函数中，那么这样真的更好吗？这种方法的好处是，客户端代码无须知道它内部是否正在使用 actor，在这种情况下，actor 已经成为一个实现细节，而不是 API 内的一部分了。再次使用这个 API 时，我们刚好可以利用这种函数调用，这意味着当我们将此函数调用与其他函数调用一起嵌套使用时，代码看起来会更加一致、更加干净。随着代码库的扩大，actor 变得越来越复杂，因此这样的隔离对于保持系统的可维护性至关重要。

那么是否应该在任何情况下都这样做呢？所有的 actor 都应该用 API 来封装吗？这样应用程序就无须在意自己是否在和 actor 打交道了。答案明显是“不”。比如我们用 Actor 模型构建的调度系统的调度库就没必要进行这样的封装，相反地，这样做会造成不必要

的复杂性。实际上这样做会使 actor 之间的交互变得更难，当使用 Actor 模型时，我们应该面对 actor，而不应该试图隐藏这个事实。只有在系统的边界上——在那里我们会想要转换到不同的计算模型上——才应该创造这种隔离。

何时适合使用 Actor 模型

上文中已经确定，有的时候使用actor更适合，有的时候则应该选择使用完整的Actor模型。而且使用它们时需要注意上下文，小心谨慎。但是什么时候是使用它们的正确时机呢？什么时候应该使用独立的 actor，什么时候应该建立一个完整的 Actor 系统呢？当然，不可能有明确的规则告诉我们应该在何种情况下使用何种方案。每种情况都是独一无二的，不过，有一些指导建议可以作为实际操作时的参考。

第一个很明显需要选择 actor 的情况是，对高并发有严格要求的同时又需要维护某种状态。毕竟维持并发状态是 Actor 模型里面最基本的属性。

另一个明显应该选择 actor 的情况是，构建有限状态机的时候。因为 actor 的自然结构就很适合用于这种场景。另外，如果只是处理一个有限状态机，一个 actor 就可以完成了，但是如果要处理多个有限状态机，而且还可能彼此交互，那么就应该考虑使用 Actor 模型。

还有一个更微妙的适合用 actor 的情况是，需要高并发，同时也需要很小心地管理并发。例如，我们需要确保特定的一组操作可以与系统中的某些操作并发运行，但不能与系统中其他操作并发运行。在调度示例中，这可能意味着我们希望多个人能够同时修改系统中的项目，但不希望这些人同时修改同一个项目。这种情况下刚好可以利用 actor 提供的单线程工作模式，这些人可以各自独立操作，但是系统中对于特定项目进行的修改只能在一个 actor 中操作。

结论

Actor 模型是一个强大的工具，像其他所有强大的工具一样，我们需要了解它并细心控制它才能从中获得最大的收获。不应盲目地假设 Actor 模型在任何情况下都是最合适的。但是，在对它有一个扎实的理解并开始考虑如何应用及何时应用它时，会发现它实际上可以应用于很多不同的场景下。因为它很自然地反映了现实世界的异步性质，因此在很多情况下都是一个很好的用于建模的工具。掌握 Actor 模型之后，我们可以摆脱全局一致性的束缚，并且确信没有任何事情是绝对瞬间发生的，都是相对于观察者而言的。

Actor 模型是一种用于组织应用程序功能的不同范式，而且具有许多优点。本章中提到

的基础概念将在本书的其他部分进一步介绍，以让大家学会如何构建优秀的 actor，并发挥该模型的最大优势。

Actor 模型中的很多细节会让初学者觉得不太寻常甚至感到束缚。但是这种限制是有一定道理的，而且 Actor 模型带来的好处会远远弥补当初为了掌握它所付出的努力。

第 2 章

Akka简介

在本章中，我们将介绍开源库 Akka。大家可能已经知道 Akka 是什么了，但也许并不知道关于它的来龙去脉。

Akka 是一个工具包，它可以帮助我们实现本书后面章节中涉及的所有模式，并且允许我们直接在自己的实际项目中使用这些技术。

Akka 是什么

官方网站上是这样介绍 Akka 的："Akka 是在 Java 虚拟机（JVM）上构建高并发、分布式、弹性消息驱动应用的开源工具包。"

Akka 支持多种编程模式，并且强调 Erlang 风格的基于 actor 的并发。

以上便是关于 Akka 的简单介绍，但它并不是仅有这些特性。下面让我们更深入地来了解它吧。

Akka 是开源的

Akka 是根据 Apache 2 许可证（一种公认的开源许可证）发布的开源项目，可以同时在其他开源或商业函数库及应用程序中被自由使用和扩展。

虽然也可以在 Java 中使用 Akka，但是它本身是用 Scala 编写的，所以自然从 Scala 语言中获得了许多特性，包括强类型安全、高性能、低内存占用，以及与所有 JVM 库和语言兼容。Akka 中大量使用的 Scala 语言的一个关键特性是，不可变数据结构。Akka 在其消息传递协议中经常使用 Scala，事实上，Java 开发者为了节省时间而特意使用 Scala 编写消息的情况并不罕见。

甚至还有专门为了方便 Clojure 使用 Akka 而设计的适配器工具包，如果不了解 Clojure，可以查阅一下相关资料，其实它是 JVM 上的一个 Lisp 语言变种。

Akka 正在蓬勃发展

Akka 还在不断更新迭代中，官方的更新频率基本保持在每几个月至少会发布一个小版本的水平上，函数库和辅助插件的生态系统要更加活跃。

还有一些人正在努力把 Akka 框架转移到 .NET 平台以及 JavaScript 上。

Akka 的更新频率非常快，但官方会继续长期支持稳定版本。Akka 大版本之间的升级过程都是非常顺利的，从而可以让使用 Akka 的项目以最小的风险进行定期升级。

Akka 是为分布式设计的

Akka 与许多其他版本的 Actor 模型一样，不仅适用于单台多核的计算机系统，还适用于集群环境。因此，Akka 的设计初衷就是为本地开发和分布式开发提供一样的编程模型——当需要把代码部署在集群环境中时，其开发方式和本地开发方式几乎一样，因此可以轻松地在本地进行代码开发和测试，然后部署到分布式的集群环境中。

在 Akka 系统里面，人们更习惯将交互行为建模成消息通信机制，与之相对的是远程过程调用（RPC）系统，它们把交互行为建模成过程调用。这是一个重要的区别，我们将在后面的章节中详细讨论。

虽然可以用 Akka 编写专门用于集群环境下的分布式代码（例如通过监听节点加入或离开集群的事件），但是 actor 内部的实际逻辑不会改变。无论是和本地还是远程的 actor 交互，它们的通信机制、交付保证、故障处理和其他概念都保持不变。

Akka 提供了构建响应式系统需要的关键特性。这类系统是指严格遵守响应式声明中的基本特性的系统（可以通过 *http://www.reactivemanifesto.org/* 来查看）。该声明描述了响应式程序都应遵循的特征：高响应性、高容错性、高可伸缩性、消息驱动。其中消息驱动正是 Akka 所拥有的关键特性，Akka 通过它支持其他特性。

在 Akka 系统中，一个 actor 可以是本地的，也可以是远程的。如果发送和接收 actor 处在同一个 JVM 中，那么它相对于这些 actor 就算是本地的。

如果一个 actor 被确定属于本地，Akka 便会在消息交付过程中进行一些优化（比如不需要序列化或网络调用），但代码并不会与远程 actor 交互的代码有任何不同。

这就是我们所说的“为分布式而设计的”：本地操作和分布式操作之间的代码没有任何变化。

Akka 组件

Akka 不仅包含其本身的核心模块，还包含一组丰富的可选附加库，可以自定义应用到项目中的 Akka 的功能。大多数 Akka 组件都是开源的，但也有商业插件可用。

Lightbend 为 Akka 以及相关的响应式平台提供了商业支持和保障，还有许多机构围绕 Akka 提供开发和咨询服务。这种商业生态系统进一步降低了机构使用 Akka 的风险，同时也不会限制想要使用 Akka 的爱好者或开发者。

Akka actor

Akka 库的主要组件是 actor，它是 Akka 的基础构建块，支持 Actor 模型的所有属性，但只有在单个 JVM 分布式和集群支持上是可选组件。

Akka actor 实现了状态不共享的、基于异步消息传递机制的 Actor 模型。它们还支持一个成熟的错误处理结构，稍后将会介绍。

Akka API 有意限制对 actor 的访问，actor 之间的通信只能通过消息传递来实现。因为无法以同步方式在 actor 中调用方法，所以在 API 和时间上，actor 之间都保持着解耦的状态。Akka 是通过使用 `ActorRef` 来实现上述机制的。当一个 actor 的实例被创建时，只返回一个 `ActorRef`。`ActorRef` 是对真实 actor 的一个封装，将该 actor 与其余代码隔离开。actor 的所有通信必须通过 `ActorRef`，并且 `ActorRef` 不提供任何对其包装 actor 的访问权限。

这与常规的面向对象方法的调用不同，常规的面向对象方法通过阻塞调用者将控制传递给被调用对象，然后在被调用对象返回时才恢复调用者以继续执行操作，而这里给 actor 的消息是通过 `ActorRef` 发送的，并且调用者不会被阻塞，而是立即执行后续操作。然后接收端的 actor 会一次一条地处理接收到的消息，就像和调用者在完全不同的线程环境中一样，不受调用者的影响。事实上正是 actor 一次只处理一条消息的机制才使 actor 的单线程模式可行。在 Akka actor 内部，只要不主动破坏 actor 的单线程模式，就可以确保只有一个线程将会修改该 actor 的状态。后面我们将讨论破坏单线程模式的方法，以及避免这样做的原因。

Akka 中的 Actor 模型的消息驱动本质上是通过使用 `ActorRef` 来实现的。`ActorRef` 提供了许多方法允许我们向里面的 actor 发送消息，这些消息通常是不可变的 case 类的形式。

即使没有任何原因阻止我们向一个 actor 发送可变的消息，但这依然是一种不好的做法，因为这样做可能会打破 actor 的单线程模式。

在 Akka 中，actor 的行为变化是通过使用 `become` 来实现的。任何 actor 都可以调用 `context.become` 来为下一个消息提供一个新的行为。也可以使用此技术来改变 actor 的状态。但是，actor 也可以通过使用可变字段和类参数来维护和更改状态。稍后将更详细地讨论改变状态和行为的技术。

子 actor

Akka 中的 actor 也可以创建子 actor。actor 有多种可用的工厂方法允许其创建子 actor。系统中的任何 actor 都可以访问其父级 actor 和子级 actor，并可以自由地向它们发送消息。这正是使 Akka 支持监督机制的原因，因为父级 actor 会自动监督它们的子 actor。

Akka 中的监督机制意味着消息传递的结构可以专注于所谓的“快乐路径”，因为错误消息具有完全不同的传播路径。

当一个 actor 遇到错误时，调用方并不会发觉这个错误——毕竟，调用方可能都已经不存在了，或者在另一台完全不同的机器上，因此返回错误给调用方并不一定会有意义。相反，Akka 会将 actor 的错误消息传递给它的监督者 actor，监督者 actor 一般是开始时启动它的那个 actor。这些监督者 actor 使用一个特殊的方法来处理异常，并通知 actor 抛出异常以及下一步做什么——忽略错误或重新启动（各种选项）。

这正是 Akka 和“Let it Crash”产生关联的地方。换句话说，在 Akka 里，发生错误的 actor 简单地“崩溃”是可以接受的。我们不会试图阻止 actor 发生故障，相反，我们会在 actor 发生故障时去恰当地处理它。如果一个 actor 发生崩溃，可以采取某些行动来恢复该 actor。这些行动包括停止 actor、忽略故障并继续处理其他消息、重启 actor。当重启 actor 时，我们在对用户透明的情况下将其替换为该 actor 的新副本。因为通过 `ActorRef` 处理通信，所以 actor 的客户端无须知道 actor 已经发生崩溃。从它们的角度来看，没有任何变化，它们继续指向同一个 `ActorRef`，虽然支持 `ActorRef` 的 actor 已经被替换。这样可以使整个系统具有高容错性，错误被隔离在失败的 actor 内部，不会被进一步传播。

图 2-1 显示了 JVM 中一个 actor 系统内的三个 actor。任何 actor 都可以与其他两个 actor 进行双向通信（此图中只显示了两个可能的路由）。

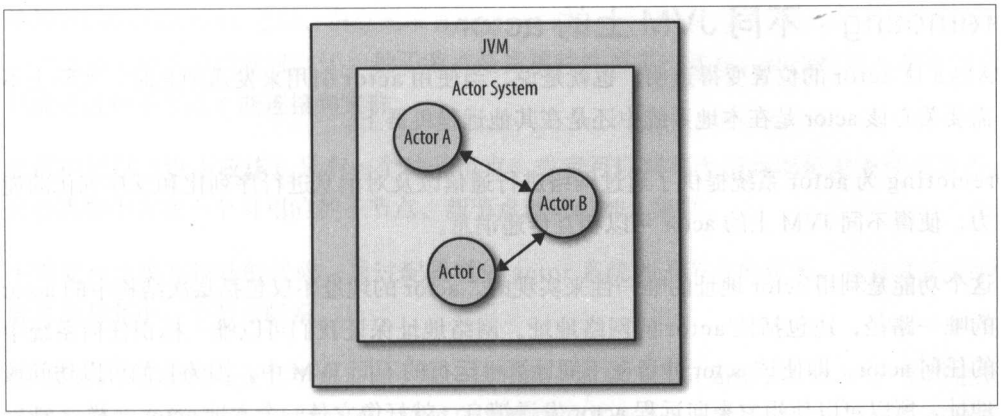


图2-1　JVM内的actor

除了常见的点对点消息传递机制，Akka 还提供了一个一对多的消息传递机制，称为事件总线（event bus）。事件总线允许单个 actor 发布消息，然后其他 actor 根据其类型订阅该消息。这样发送端的 actor 与接收端的 actor 可以完全解耦，这在某种情况下是非常有用的。

需要注意的是，事件总线仅对本地消息传递机制做出优化，对于在分布式环境中使用 Akka，发布 / 订阅模式对多节点机制做了等效的优化。

图 2-2 同样显示了三个 actor，但这次它们之间不是直接通信，而是与事件总线进行双向通信，从而改善了解耦情况。

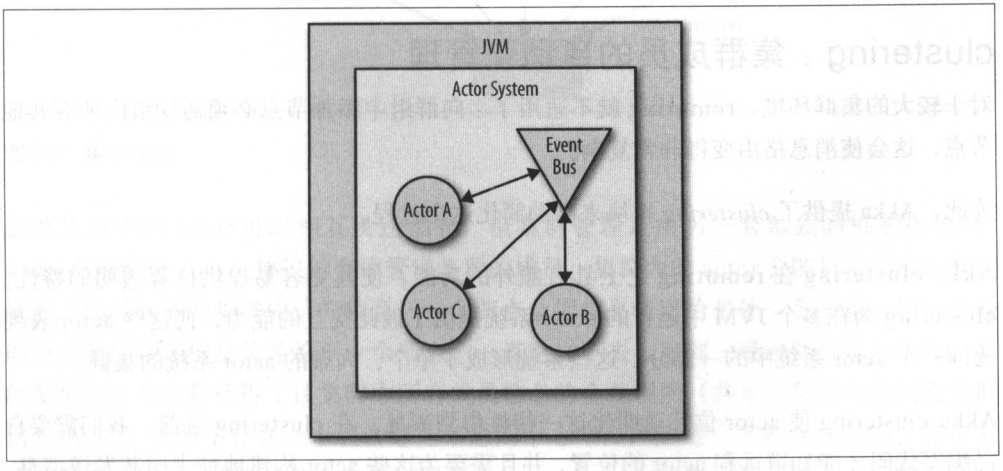


图2-2　actor和事件总线

remoting：不同 JVM 上的 actor

Akka 让 actor 的位置变得透明，也就是说，当使用 actor 引用来发送消息时，实际上不需要关心该 actor 是在本地系统中还是在其他远程机器上。

remoting 为 actor 系统提供了通过网络进行通信以及对消息进行序列化和反序列化的能力，使得不同 JVM 上的 actor 可以相互传递消息。

这个功能是利用 actor 地址的唯一性来实现的。actor 的地址不仅包括层次结构中的 actor 的唯一路径，还包括该 actor 的网络地址。网络地址保证我们可以唯一标识任何系统中的任何 actor，即使该 actor 驻留在不同计算机运行的不同 JVM 中。因为我们可以访问该地址，所以可以使用它来向远程 actor 发送消息，就好像它是一个本地 actor 一样。Akka 平台将为我们处理交付机制，包括序列化和发送消息。

序列化机制是可插拔的，通信协议也是如此，所以有很多选择可用。在项目前期，Akka 中的默认机制通常足够满足需求，但随着应用程序的日益复杂，可能需要考虑使用不同的序列化机制或通信协议来满足新的业务需求。

Akka 中的 remoting 只能通过配置来添加：不需要更改代码（也可以编写代码显式执行 remoting）。

每个节点必须知道其他节点的存在，并且必须具有其网络地址才能使用 remoting，因为消息传递是点对点和显式的。Akka Remoting 不包括任何发现机制，发现机制是 Akka clustering 的功能。

clustering：集群成员的自动化管理

对于较大的集群环境，remoting 就不适用了：向群组中添加节点必须通知组内所有其他节点，这会使消息路由变得非常复杂。

为此，Akka 提供了 *clustering* 模块来帮助简化这个过程。

Akka clustering 在 remoting 之上具有额外的功能，使其更容易提供位置透明的特性。clustering 为在多个 JVM 中运行的 actor 系统提供了彼此交互的能力，使这些 actor 表现为同一个 actor 系统中的一部分。这些系统形成了单个、内聚的 actor 系统的集群。

Akka clustering 使 actor 位置透明化这一特性得到彰显。在 clustering 之前，我们需要自己编写代码才能知道远程 actor 的位置，并且需要为这些 actor 构建地址来向其发送消息。更棘手的问题是故障的恢复，如果正在通信的远程 actor 节点不可用或失败，则需要自己处理启动另一个新节点并创建与该节点的连接。

使用 Akka clustering 之后，节点无须直接了解网络上所有其他节点的信息，只需要知道一个或多个种子节点就可以了。种子节点是指被特殊指定的节点，那些想加入的新节点只能通过种子节点才能连接到集群。

集群中可以（但不应该）只有一个种子节点，或者可以将每个节点都指定为种子节点。只要集群中存在一个可用的种子节点，新节点就可以加入集群。

不需要自己编写特殊的代码：通过配置通知 actor 系统种子节点的位置，系统在启动时会自动与其中一个节点联系。

然后该新节点会经历正常生命周期中都会涉及的一系列步骤（如图 2-3 所示），直到它成为集群中的完整成员，此时它不仅能够向集群中的其他 actor 发送消息，还可以在自己的 actor 系统或集群的另一个节点上启动新的 actor。实际上，该集群变成了一个单一的虚拟 actor 系统。

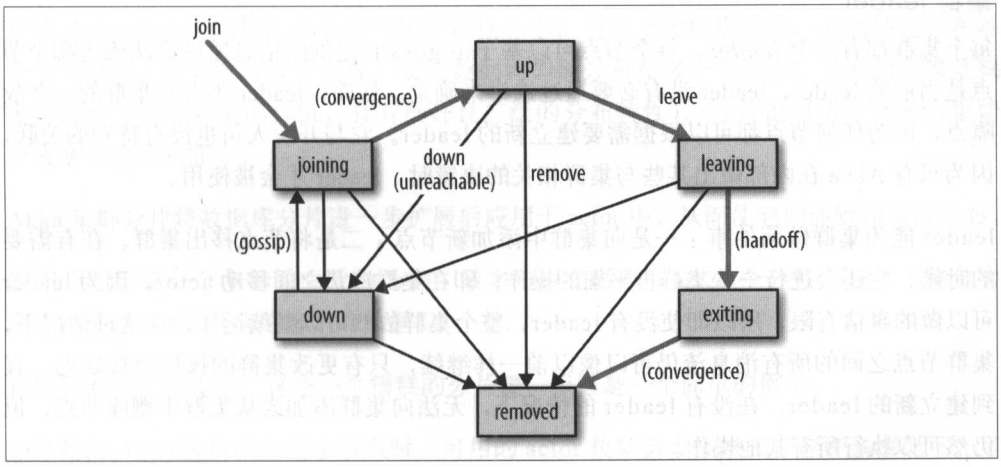


图2-3　集群状态

虽然集群中的 actor 可以相互发送消息，但集群管理是由另一套单独的机制控制的。actor 系统使用 *gossip* 协议的变体管理集群的成员。集群内的 actor 会随机配对然后互相交换它们所知道的集群成员的信息，一个节点会把信息传递给另外一个节点，然后另外的节点会继续把信息传递给下一个节点，一直这样下去，直到一段时间后发生收敛。收敛是指所有节点都获得了该集群中所有成员信息的全貌视图（并且它们具有相同的集群信息结构），但是不存在一个包含这些信息的中心节点。这是一个鲁棒性强的、高可容错的系统，可用于许多其他分布式应用程序，如 Amazon 的 Dynamo[1] 和 Basho 的 Riak[2]。

1　*http://amzn.to/2gW7rVW*
2　*http://basho.com/products*

实际上，矢量时钟并不是跟着世界标准时间在走，而是为了确定分布式系统中事件的局部排序方法，然后将通过 gossip 协议接收的消息调整为集群的准确状态。

当集群发生与生命周期相关的事件时，会自动发送特定的事件消息——例如，新节点加入或现有节点离开集群。使用集群时其实并不需要监听这些事件，但如果想要实现更细粒度的控制或监视，也可以监听事件。

对于被确定为从集群中丢失的节点，可以使用 Phi 应计故障检测器来确定节点何时变得不可达，并且最终下线。当然，节点可以故意地离开集群，例如在缩减集群规模或集群关闭期间。

有关 Phi 应计故障检测方法的说明，请参阅论文 *The φ Accrual Failure Detector*（*http://bit.ly/phi-accrual*）。

集群 leader

每个集群都有一个 *leader*。每个节点可以基于由 gossip 提供的信息通过算法确定哪个节点是当前的 leader，leader 没有必要通过选举来确定。然而，leader 不会是集群的一个故障点，因为任何节点都可以根据需要建立新的 leader。它与开发人员也没有特别的关联，因为只有 Akka 在内部做出某些与集群相关的决策时，leader 才会被使用。

leader 能为集群做两件事：一是向集群中添加新节点；二是将节点移出集群。在有需要的时候，它还会进行一些集群再平衡的操作，即在集群成员之间移动 actor。因为 leader 可以做的事情有限，所以即使没有 leader，整个集群依然可以继续运作。在这种情况下，集群节点之间的所有消息流仍可以像以前一样继续，只有更改集群的操作将被延迟，直到建立新的 leader。在没有 leader 的情况下，无法向集群添加或从集群中删除节点，但仍然可以执行所有其他操作。

分布式系统的一个潜在问题是分割（partitioning）。在集群中，如果一个节点意外终止，其他节点只需要简单接管它的功能即可。然而，如果一组节点突然与集群断开连接——例如，发生网络故障——但这组节点并没有被终止运行，那么问题就出现了。这部分所谓的幸存者节点组会继续运行下去，但到底哪部分才应该被算作是幸存者呢？两部分节点组完全被隔离后，它们可能会自己组建一个新的集群，最终可能导致脑裂（split-brain）的情况出现，形成两个集群，破坏集群单例的唯一性。

有几种解决方案可以用于处理这种情况。最简单的是禁止 actor 自动下线（autodowning）。当启用自动下线时，如果节点不可访问，系统将在一段时间后自动将其从集群中移除。这个方案理论上来看是可行的，但在实际操作中有很大问题：导致脑裂。一个或多个被移除的节点可以形成它们自己的新集群，从而引发问题，然而通过禁用自动下线可以避

免这种情况。当节点变得不可用时，需要人为干预才能从集群中移除节点，因为无法达到集群聚合所需要的条件，所以这样不会形成新的集群。通常在生产环境中不提倡启用自动下线功能。

但是，如果必须使用自动下线功能，也有其他可行的方法可以避免上述问题，比如可以限制形成集群的最小规模（这是可以配置的）。在这种情况下，如果单个节点断开连接，集群则不能形成，因为没有足够的成员。选择正确的集群最小规模要视情况而定，并没有一个确切的标准，需要根据当前的实际业务进行调整。

Lightbend 提供了一个智能脑裂解析器，该仪器作为一个商业化的附加组件可以帮助解决上述问题。它使用更成熟的技术来确定是否已发生集群分割（partitioning），然后采取适当的措施。

集群分片

分片（sharding）的想法最初被应用于数据库中，当单个数据集过大，无法被包容在单个节点中时，需要进行分片处理。要使数据均匀地分布在多个节点上，关键的步骤是确定分片键（数据的某部分值，具有良好且广泛的分布特性）。分片键可用于将数据集切分成多个较小的部分。

Akka 集群分片将数据库分片进一步扩展后应用于 actor 中，从而使它们能够跨集群分布。

以用户姓氏第一个字母的分片键为例：最多可以把数据分成 26 个分片，分片 1 将包含所有姓名以“A”开头的客户的数据。

在实际的生产环境中，这是一个糟糕的分片键，但这是一个简单的例子。

当开始向 Akka 集群添加更多节点时，可用的 actor 数量会大幅增长，在许多情况下，可以像数据库分片那样对集群进行分片。

Akka 集群分片结合了传统分片的概念，同时对它进行了扩展，所以可以在集群中对 actor 进行分片操作。

每个节点都可以被分配给一个分片区域（shard region），并且每个区域至少会被分配一个 actor 节点（实际生产环境中应该会多于一个）。

专门用于集群分片环境中的消息会被包裹一层特殊的包层，这层包层包含的值可以用来确定该消息需要投递到哪个分片区域。解析到特定分片区域的所有消息仅传递到特定的区域，从而允许相关 actor 中包含的状态仅存储在特定区域的节点上。

在前面的简单示例中，如果有一个分片区域专门给所有名字以“A”开头的客户使用，

则可以用客户姓氏作为值来解析并得到该客户所属的那个分片区域，将消息传递到正确的节点。

具有集群分片的分布式域

集群分片的一个显著优点是，区域内 actor 需要持久化的状态，可以被限制到特定分片区域内的节点上，不需要在整个集群上被复制或共享。例如，假设正在为客户保存账户余额，可以通过存储每个更改余额的消息来记录余额，然后将这些更改记录保存到磁盘以进行持久化。这样，当记录此状态的 actor 重新启动时，只读取那些持久化的更改记录就可以得到客户的正确账户余额了。

通过使用 Akka clustering 和集群分片的功能，可以构建一个使用分布式域（distributed domain)的系统。这是一种在集群分片的 actor 系统中通过一个 actor 保持某个域(如客户)实例状态的技术手段，稍后将深入探讨这一模式。

假设正在讨论的 actor 是针对姓名以“A”开头的客户，则相关的日志仅需要保存在特定分片区域的节点上——因为我们可以保证该客户的 actor 仅在该分片区域中的某一个节点上启动——与常规的集群情况正好相反，在集群中的任何地方都可以启动一个 actor。应用此技术需要特别注意，因为该技术明显限制了集群的灵活性。

集群单例

集群的另一个重要的点是集群单例（cluster singleton)，即在集群中始终有且只有一个特定的 actor 实例。它在集群中的位置并不重要，重要的是只能有一个。

在这种情况下，Akka 提供了一种特殊的方法来确保每个节点都能启动单例，但在某个特殊时刻只能有一个节点这样做。如果由于某种原因使当前单例的 actor 失效，则相同节点或另一个节点将再次启动替代它，尽可能地确保有单例可用。每个需要发送消息到单例的节点都有一个代理，它能保证该单例无论在集群的什么位置，都可以收到消息。

当然，集群单例有和其他单例机制一样的缺点，只是系统的故障转移机制可以帮助其克服大部分缺点，但是仍然可能遇到性能瓶颈，因此必须小心使用集群单例。

Akka HTTP

受 Spray HTTP 库的启发，Akka HTTP 更深入地集成了 Akka 和 Akka Streams，从而取代了 Spray HTTP。Akka HTTP 提供了一种在 Akka 之上构建 HTTP API 的方法，这也是让 Akka 对外提供 HTTP 接口的推荐方法。

TestKit

任何一个好的工具包的核心功能都是测试我们使用该工具包生成代码的能力，Akka 也不例外。异步和并发程序是非常难以测试的，但 TestKit 提供的功能可以轻松、完整地实现该测试。而且我们甚至可以先写测试代码，不会遇到开发异步应用常遇到的困难。

TestKit 克服了并发性或分布式系统最棘手的问题之一：如何反复可靠地测试这样的系统？

并发的性质导致各个操作之间的确切执行顺序是未知的，分布式系统也是如此。分布式系统也是并发的，但是增加了涉及多个物理系统的复杂性，还需要把诸如网络通信和序列化 / 反序列化等元素引入测试方程中。

TestKit 可以通过两种方式进行测试。最简单的是提供一种方法来测试临时的 actor——只针对测试的范围——允许以完全同步和确定的方式访问 actor，这使得 actor 不会比其他代码更难测试。然而，这种方法具有隐藏问题甚至产生问题的可能性。一个完全同步和确定的 actor 并不能代表实际生产环境中的情况，它可以以允许测试通过或失败的方式去改变 actor 的行为，否则就表示还是有 actor 以异步的方式在工作。

TestKit 的第二个功能是提供验证 actor 在非确定性和异步模式下发送及接收消息的方法：通过一个简单的方法来存根（stub）或伪造 actor，同时提供相应的方法来断言（assert）某些消息已在特定的超时时段内被接收，而无须指定实际操作发生的先后顺序。

最后一个附加组件是 multi-JVM 测试，其实它不是 Akka TestKit 的一部分，但是可以很好地与之结合使用。这是一种简单的测试手段，用于启动 JVM 的一个全新实例，模拟节点网络，并可以验证隔离的 actor 系统之间的交互。此功能由 Akka 团队开发，用于在接近生产环境的设置中测试跨多个虚拟机（VM）甚至多个物理节点的 actor。

contrib

Akka 提供了一个名为“contrib”的库，其中包含许多不同的工具，如用于限制消息、聚合消息等的有用工具，值得去探索发现。

Akka OSGi

OSGi 模型有很多优秀的特性，因此成为 Akka 主机环境的最佳选择，并且该模型提供了许多特定的支持，允许 OSGi 生命周期在正确的时间使用 Akka 支持来初始化 actor 系统内的模块。

Akka HTTP

Akka HTTP 提供了将 actor 对外暴露成一个 REST 服务端点的方法，并构建了 RESTful Web 服务。

Akka Streams

Akka Streams 提供了一个更高级别的 API 来与 actor 进行交互，同时提供自动处理“背压（back pressure）”的机制（我们将在后面详细讨论）。

Streams 提供了一种构建基于 actor 的流和图形的复杂结构，可以使用这些结构的同时结合我们熟悉的特定领域的语言（DSL）来处理和转换数据。

这些 Streams 遵循 Reactive Streams 标准。

Akka 实现的 Actor 模型

前文提到，Akka 在 JVM 之上提供了 Actor 模型。下面来看一下 Akka 是如何提供 Actor 模型的相关属性的。

Actor 模型中的 Akka actor

Actor 模型中的基本计算单位是 actor，而 Akka 直接提供了 actor，还有与 actor 及 actor 专业化相关的一些特性。在即将到来的 Akka 版本中，类型化（typed）actor 实际上会删除名为“Actor”的特性，用特定的特征代替它来定义 actor 的行为，从而在语义上产生同样的效果。

如 Hewitt 所描述的，Akka 的 actor 是可以执行计算操作的。

一个简单的 actor 的代码如下。

```
import akka.actor.Actor

class DemoActor extends Actor {
  def receive = {
    case _ => println("Received a message")
  }
}
```

消息传递

Actor 模型指定消息传递应该是 actor 进行通信的唯一方式，并且所有处理操作都应该是为了响应此消息传递才发生的。

在 Akka 中，消息是 actor 之间或 actor 与外界进行交互的唯一手段。消息通过邮箱的非阻塞队列传递给其他 actor。actor 本身的对象引用并不会被直接使用，而是用一个名为 `ActorRef` 的中间件。此队列通常是先进先出的，如果需要也可以换成其他形式。

Akka 中有三个消息传递机制，下面分别来介绍。

tell

发送消息的首选机制是 tell，有时称为“bang”，基于方法的简写方式为“！”。

通过使用“！”运算符可以指定一个“tell”，代码如下。

```
projectsActor ! List
Which is the equivalent of
projectsActor.tell(List)
```

（假设 List 在此处是一个 case 对象。）

tell 方法是 Actor 模型中经典的“fire and forget”消息机制：它既不阻止也不等待任何响应。

ask

在使用 ask 方法时需要额外小心，因为使用 ask 很容易编写出阻塞型代码，这将大幅降低异步系统的性能。

ask 的简写形式是“？”，示例如下。

```
projectsActor ? List
```

执行 ask 操作后会有响应返回，这个响应可能需要等待一段时间后才会被捕获，因为被请求的 actor 并不一定会立即响应。

发布 / 订阅

Akka 发送消息的最后一种方式是前文中提到过的采用事件总线的方式。消息的发送者通过引用事件总线来发布消息。消息的接收者必须先订阅该类型的消息，然后才能接收其他 actor 发布的同类型的所有消息。两个互相通信的 actor 并不直接知道对方的存在，并且发送方引用都不可用于以这种方式接收到的消息。

actor 系统

根据 Actor 模型的定义，单个 actor 根本不算是 actor。Akka 提出了 actor 系统的概念，即容易实现消息交换的 actor 群组，无论是在单个 JVM 中运行还是跨越多个 JVM 运行。

虽然不同 actor 系统中的 actor 也可以进行通信，但这并不常见。

actor 系统还可以创建或定位 actor 的工具，代码如下。

```
import akka.actor.{Actor, ActorSystem, Props}

val system = ActorSystem("demo")

val actor = system.actorOf(Props[DemoActor])
```

创建新的 actor

Akka 中的 actor 可以从系统外部被创建，或者被其他 actor 创建，只要满足 Hewitt 的要求，新的 actor 就可以创建子 actor。

改变行为

Akka 中的 actor 可以转换它们响应消息的行为，新的行为可以处理完全不同的消息集。Akka 通过 untyped actor 执行 become 和 unbecome 操作进而实现这种转换。Akka Typed 是 actor 的未来版本，稍后将详细讨论，每次处理一个消息之后，Akka Typed 会返回处理下一个消息的新行为。

Akka 进一步提供了一个方便的 DSL，专门用于创建有限状态机的 actor，这是一个改变行为的常见用例。

Akka Typed 项目

在 Akka 2.4 中，一个实验性项目 Akka Typed 被添加到 Akka 的主要分支内。

这个项目实际上是尝试结合 Scala 的强类型特性与 actor receive 方法特性的最新成果。目前的 receive 方法实质上可以看作一个偏函数，它接受任何类型的参数并返回 unit 类型的结果。

这是因为任何消息类型都可以被发送给 actor，尽管已经开始尝试引入类型约束，但类型系统中没有任何事情可以阻止我们这么做。

Akka Typed 有一个简单的前提：一个 actor 的所有参数都是围绕它的行为的，并且该行为可以具有一个类型属性。该类型指定 actor 行为可以有效处理的消息集合，除此之外

不能处理其他类型的消息，那么为什么不从一开始就阻止那些不能被处理的消息发送过来呢？

Akka Typed 也消除了 Actor 特性本身，而不是提供几种类型来直接声明行为以及它们接收的消息类型。

Akka Typed 更严格地遵循 Actor 模型：生命周期内的事件由消息表示，从而取代了生命周期方法。

在这个新版本的 Akka 中，整个 actor 系统也被类型化了。创建一个 actor 系统提供了一个构造函数，它为顶层 actor 提供了一个 Props 对象。

Akka Typed 仍然是实验性的，但它已经显示出了巨大的可能性。

结论

Akka 是 Actor 模型的一个全面实现，旨在与现有的 JVM 语言配合使用，使我们能够充分利用正在使用的语言的 actor。

如果还没有使用基于 JVM 的语言，那么当有机会使用 JVM 上庞大而成熟的系统库时，Akka 仍然可以展现它的强大吸引力。

本章介绍了将在后面章节中讨论的模型及其实现。在第 3 章中，我们将开始讨论可以充分利用 Akka 的 actor 来实现的架构及其设计方法。

第 3 章

分布式领域驱动设计

当探索 Akka 提供的工具和技术时，我们需要将所有的概念“绑”在一起。我们需要一套指导原则，它不仅能帮助设计 actor，还能指导设计组成整个系统的其他组件。当了解了诸如 clustering 之类的新概念时，需要一种将应用程序分割成较小的子系统的方式，我们要了解这些子系统的边界以及它们如何进行交互。我们将要使用的指导原则来自领域驱动设计（DDD），当和分布式 Akka 系统结合在一起时，则要使用分布式领域驱动设计（DDDD）原则。

DDD 是由 Eric Evans 创建的一个被广泛采用的概念，本章将介绍一些相关的基础知识，如果有兴趣进一步了解 DDD，可以参考 Evans 的书[1]或 Vaughn Vernon 等人编写的扩展书籍[2]。

DDD 概述

DDD 是一套关于软件架构的指导原则。DDD 原则并不是革命性的，事实上，人们第一次听说该原则时，共同的反应是：“好吧，这很明显。”并不是因为原则本身多么强大，而要看它是如何被应用和组合的。在整个项目中始终如一地应用 DDD 可以让项目从烦琐和难以理解的状态转变为优雅且相当有用的状态。

DDD 中最重要的概念是域模型（Domain Model）。“域”是构成尝试建模的业务或领域的一组要求、约束和概念。在 DDD 中，我们专注于涉及的业务领域，并围绕它构建软件系统模型。我们试图以反映现实世界及其运作方式的方式对软件系统建模，希望模型能反映建模领域的真实性。这个过程还需要与该领域的专家进行有效的对话交流。这些

1 Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of So ware.* Boston: Addison-Wesley, 2003.

2 Vernon, Vaughn. *Implementing Domain-Driven Design.* Boston: Addison-Wesley, 2013.

专家是非常了解该领域的人，但可能并不精通计算机及相关技术。他们包括律师、营销人员、客户服务人员、业务经理或任何其他领域的专家。这意味着不仅在对话中，甚至在代码中都要使用能让这些人理解的语言。我们开发的新语言应该是开发人员和领域专家共享的，被称为通用语言（ubiquitous language）。

建立通用语言不仅帮忙简化了技术开发人员之间的沟通交流，而且也使开发人员与领域专家之间的沟通变得简单很多。通用语言可以让我们在讨论软件系统的时候，用非开发者也可以理解的方式引用应用程序中的操作和对象，这也能让那些领域专家更加觉得自己参与到了这个系统的开发过程中。当使用他们理解的语言解释模型时，他们能够指出使用该语言介绍的模型中的缺陷。这些缺陷通常会反映出开发人员对该领域的错误理解，而这些错误往往会悄悄进入软件系统里面。这种通用语言是软件开发过程中一个非常有用的工具。

这种通用语言在不同的领域会有不同的意义。可用的操作以及它们在领域内的交互方式可能并不总是相同的，每个概念都是和特定的上下文进行关联的。在一个特定的上下文中，它代表一种特定的意义，但当离开这个上下文而到了另外一个领域时，它代表的意义可能就完全变了。

这里的关键点是，我们需要意识到业务涉及的领域并不是一成不变的，它就像是一个会随时间变化的流体，有时是业务规则发生了变化，有时是我们对这些规则的理解发生了变化。我们需要为这些变化做好准备，随时准备改变域模型以适应这些变化。如果我们建立一个模型，期望它处理所有的情况，并且永远不会改变，那么我们注定要失败。

DDD 专注于构建能够持续发展的模型。我们不是要建立一个自始至终都不变的模型，模型将在将来的某一时间点面临失败，只有改变它才能适应新的业务。因此，我们需要以随时可以改变它的方式去构建它，而且改变的成本要尽可能低。DDD 为我们提供了一套工具，可以以很低的成本去改变模型。

DDD 的好处

削弱系统并防止其进一步发展的最糟糕的方法之一是在系统内不同区域之间建立过多的耦合，特别是在系统的一个基础架构和系统的一个业务域模块之间建立过多的耦合，这样就更难去优化调整系统了。DDD 会帮助我们区分系统的业务域部分和基础架构部分，同时帮助我们在它们周围创造正确的抽象层，这样就不会触及边界。

来看一个非常简单的例子：假设公司多年来一直使用某个特定的电子邮件库，这个库一直运行良好，但是有一天业务方发现了一个他们非常喜欢的新的电子邮件服务，这个新服务提供了很多之前的旧服务没有的工具，可以生成很多用户相关指标以及数据。业务

方要求将系统转换为使用这个新电子邮件服务的系统。

这时，我们开始深入系统去查看代码，然后意识到这是一个大难题。因为旧的电子邮件库已经深深集成在代码中了，几乎每个地方都有挂钩（hooks）。如果要转换为新系统，则需要在许多不同的地方修改代码。调度引擎模块需要修改，因为它使用电子邮件发送邮件通知；用户管理系统需要修改，因为它使用电子邮件发送确认邮件和邀请邮件；甚至数据库中的存储过程也触发了电子邮件。到底哪里才是尽头呢？

DDD 引入了许多有助于解决这类问题的方法。它帮助我们认识到，发送电子邮件的细节并不是用户管理模块或调度引擎需要关心的问题，这是一个基础架构问题。比如调度引擎模块需要的功能只是发送通知，而不是发送电子邮件。事实上通过何种方式发送通知并没有太大影响，所以用什么版本的电子邮件库发送邮件就更加没有影响了。

认识到所使用的语言中涉及的术语间的差异（比如通知和电子邮件的差异）只是第一步。还要认识到，管理用户以及对这些用户做出决定是完全不同的。当然，只创建一个模型来满足所有需求可能并不容易，或者基本办不到。

DDD 的目标是将较大的领域业务分解成更小、更易于管理的业务块，然后我们就可以分别对这些小的业务块建模，这样开发的系统不仅对我们来说更好理解，对领域专家来说也更好理解。可以找专家讨论调度引擎模块发送通知的功能，当然，事实上还是用电子邮件功能，但是通过使用“通知”这个术语，我们做好了在将来的某一时刻把邮件功能替换为短信功能，或者社交媒体信息功能的准备，也为下一个可能发生的大变动做好了准备。我们也可以与专家专门讨论发送电子邮件的功能以及其中涉及的内容，不需要混淆我们关心的问题，不需要把基础架构的功能逻辑牵扯到业务功能逻辑中。

DDD 组件

那么，有什么工具可以使用 DDD 去分解应用程序呢？更重要的是，这些工具又是如何与 Akka 产生关联的呢？

DDD 提供了一组可以直接在代码中使用的工具，它们将是应用程序的构建块。除了小的构建块，还有更高层次的概念可以帮助我们理解如何使用这些基础的构建块，并将它们组合起来创建更大的软件系统。

通常这些构建块会成为确定 actor 正确结构的完美候选对象。当试图决定是否值得为一个特定的概念创建新的 actor 或是否应该将其加入一个现有的 actor 中时，就可以使用 DDD 的构建块来做出决定。这样的构建块与 Akka 有很多天然的相似之处，所以可以很容易地将它们直接映射到 Actor 模型中。

下面来看具体的关于构建块的内容。

域实体

DDD 使用实体的概念来引用系统中可由键或复合键唯一标识的对象。实体是可变的，也就是说，如果实体以某种方式改变其状态，但其键值仍然保持不变，则认为它还是同一个实体，其身份（identity）并没有发生变化。

实际上，实体可以包含可变的状态且有唯一可识别的标识，可以直接映射到 Akka actor 中。actor 其实就是管理可变状态的，并且系统中的每个 actor 都可以使用其路径进行唯一标识，无论 actor 包含的数据如何。因此，我们可以自然地在系统中使用 actor，就像在那些不是基于 actor 的系统中使用实体一样，也可以认为它们是一样的。

例如，如果系统中具有 user 实体，可以将该 user 建模为 actor，代码如下。

```
class User(id: UUID) extends Actor {
  override def receive: Receive = ...
}
```

当 user actor 接收消息时，actor 的内部状态可能会改变。但是通向 actor 的路径和 user 的 ID 不会改变，它们是固定的值，这意味着 actor 总是可以通过路径或 ID 被唯一地标识，这也使得 user 成为实体。

通常，构建 actor 来表示实体的比较好的做法是使用实体 ID 作为这个 actor 的名称。我们应该尝试建模 actor 层次结构，尽可能复制域中实体的结构。

域值对象

另一方面，值对象与实体不同。值对象在其包含的属性之外没有实体那样的标识，包含相同数据的两个值对象被认为是相同的，因此不必费心地区分它们。另外，值对象是不可变的。它们必须是不可变的，如果值对象的数据改变了，那它们就变成了不同的值对象。

在 Akka 中，actor 之间传递的消息是值对象。如果遵循最佳实践，那么通常这些消息是不可变并且不可标识的。它们只是数据容器，可能包含对其他实体的引用，但是消息本身通常不是实体，还可以使用值对象作为容纳 actor 状态的容器。我们可以根据需要在不同的状态之间切换，但状态本身没有任何身份，只有当它存在于 actor 内部时，它才是可标识的。如果创建两个具有相同状态但是各自又是独一无二的 actor，那么 actor 将被视为实体，而状态将被视为值对象。

user actor 可能有一系列消息需要传递给 actor 以改变它的状态。例如，如果需要更改

user 的名称，可以使用类似 SetName 的消息，代码如下。

```
object User {
    case class SetName(firstName: String, lastName: String)
}
```

在这种情况下，SetName 消息是值对象。如果有两个 SetName 对象，其 firstName 和 lastName 的值相同，则这两个消息可以被视为相同的消息。无论发送哪一个都没关系，对用户的影响是一样的。相反，如果要在其中一个消息中更改 firstName 的值，则会变成不同的消息，将它们发送给用户将具有完全不同的效果。这里的 SetName 消息没有唯一的标识符，除了消息本身的内容，没有办法将一个消息与另一个消息区分开。这就是值对象。

用于在 actor 之间传递的消息被称为消息协议，一个好的做法是将特定类型的 actor 消息嵌入到公共协议对象中。这个对象既可以是单个 actor 的伴生对象，也可以是单独的协议对象（例如 UserProtocol）。如果希望不同类型的 actor 处理同一组消息，那么后者就非常有用。

聚合与聚合根

聚合（aggregate）是应用程序内对象的集合。聚合创建了由系统中许多不同元素组成的逻辑分组。每个聚合都会被绑定到聚合根（aggregate root）上。聚合根是聚合中的一个特殊实体，它负责管理该聚合中的其他成员。聚合的一个属性是禁止其他聚合对聚合中的任何内容持有引用。所以如果想访问聚合中的一些元素，必须通过聚合根，不能直接访问。例如有一个表示人的聚合根，该人有一个地址实体，那么无法直接访问该人的地址实体，但是可以通过访问表示该人的聚合根进而访问该人的地址实体。

聚合及其相关的根是一个比较不容易理解的概念。可能难以确定系统中的聚合是什么，更难确定什么才是正确的聚合根。通常，聚合根是系统的顶层部分，所有与系统的交互将以某种方式与聚合根进行对接（少数例外）。那么如何确定聚合根呢？

一个简单的方法是删除法（deletion）。如果在系统中选择一个特定的实体并删除它，是否会导致系统中的其他实体被删除呢？如果系统由具有地址的人员组成，删除一个地址是否会删除系统的其他部分呢？在目前这种情况下，可能并不会。另一方面，如果从系统中删除某人，则很有可能不再需要保留该人的地址，在这种情况下，该人就聚合了一个地址。但请记住，虽然通常都可以把人直接作为系统的聚合根，但并不总是这样。以保龄球记分系统为例，在这个系统中，可能有游戏和玩家的角色。玩家看起来很像是聚合根的天然候选人，玩家有分数，如果删除一个玩家，与该玩家相关的分数也会被删除。

但如果删除一个游戏会发生什么呢？大家可能会觉得删除游戏并不一定会删除玩家，但这并不准确。删除游戏虽不删除玩家，但如果一个人没有参与到任何游戏里，该人是否会被认为是一个玩家？在这种情况下，把游戏作为聚合根显然更有意义。

虽然可能会选择不合适的聚合根，但并不是必须从一开始就选择正确的聚合根，重要的是能让改变聚合根的成本保持尽可能低。

在 Akka 中，聚合根通常由父 actor 表示。当删除 / 停止父 actor 时，它的所有子节点也会随之被删除 / 停止。聚合根不是必须由顶层的 actor 表示的，有时，在顶层 actor 和实际的聚合根之间隔着一层或两层是有好处的。例如，如果某用户是聚合根，则可能需要一个位于用户上方的层，该层将是所有用户的父级。在这种情况下，该用户仍然是聚合根，但系统中有另一个组件来负责管理这些用户。引入集群分片的概念时尤其如此（见第 2 章）。

来看一个简单的例子。在调度系统中，可能有人需要被调度。我们将使用一个 Person actor 代表一个人，该 Person 可能会有一个 Schedule，这时需要（特别是在使用 Actor 模型时）使用一个 Schedule actor 来表示该日程表，代码如下所示。

```
object Schedule {
  def props = Props(new Schedule)
}

class Schedule extends Actor {
  ...
}

class Person(id: UUID) extends Actor {
  private val schedule = createSchedule()

  protected def createSchedule() = context.actorOf(Schedule.props)
}
```

可以在此示例中看到 Person actor 聚合了 Schedule actor。如果不通过 Person 是无法访问 Schedule 的，而且如果删除了 Person，那么 Schedule 也会一同被删除。这使得 Person 成为聚合根的候选人。然而，不能局限于此，需要放眼全局。Person 还可能是其他 actor 的子节点吗？这是否意味着其他节点才是聚合根的候选人？这些是在尝试查找聚合根时需要询问的问题。

仓储

仓储是开始抽象出基础架构问题的地方，它们可以在存储问题的上层创建一个抽象层。在 DDD 中使用聚合的基本方法是先转到仓储中，从该仓储获取某一个聚合，执行一些操作，然后再次保存该聚合。这听起来很像数据库，事实上，一个仓储可以是一个数据库的抽象。但不要限制自己的思维，虽然仓储可以访问数据库，但它也可以从内存、磁盘或 Web 中提取数据。事实上它可以完成所有上面提到的操作，并没有限定单个仓储不能与多个存储机制有依赖关系。

使用好仓储的关键在于理解它们是抽象层。因此，它们通常在 Scala 中被表示为 trait。该 trait 定义了如何与仓储对接，但隐藏了关于它与数据库或其他存储机制交互的所有实现细节。然后，可以创建该 trait 的基础架构的相关实现，供域代码使用。

在 Akka 中，当使用 Actor 模型时，仓储可能会变得有点棘手。仓储涉及的一般流程类似于 Akka Ask 操作：向仓储询问某个聚合的实例，对该聚合执行操作，然后指示仓储提交该更改。问题是，这违反了“Tell, Don't Ask”的原则（后面会介绍）。通常在 Akka 中，仓储会表现得稍微有些不一样：不是向仓储直接请求特定聚合的实例，而是指示仓储向该聚合发送消息。仓储就像一个“管理人员”，它是某一组 actor 的父节点。我们要通知管理员，想要一个特定的 actor 来处理一个消息，然后仓储就会负责定位该 actor 并向其传递这条消息。

还需要记住，仓储的目的是从基础架构问题中抽象出抽象层。我们使用仓储的目标通常是从数据库或其他存储机制中重新构建聚合。如果使用 actor 来表示聚合，可能需要从数据库中加载一些数据，使用这些数据创建适当的 actor，然后传递消息到该聚合。聚合本身仍然可以是域的一部分，仓储的接口也是该域的一部分。然而，仓储的实现细节却是基础架构的一部分，而不是域的一部分。

将实现细节作为基础架构的一部分来处理是很重要的。我们的目标是创建一个将域和基础架构隔离开的系统，无须关心是否使用 SQL 数据库、NoSQL 数据库、数据文件或任何其他结构。理想情况下，如果需要，我们希望能够在不同的仓储实现里面进行自由切换，这对于测试来说很有用，但随着应用程序的发展，它对于生产环境中代码的演变也很有价值。我们不想假设当前使用的数据库的实现是静态并且将一直保持不变的。相反，我们希望它随业务发展而演变。随着需求的发展，我们可以编写新的仓储并使用它们，而无须重新编写与该仓储交互的逻辑代码。

在调度域中，假设已经确定了 `Person` 是一个聚合，则可能有一个 `PersonRepository` 来管理该聚合的实例。如果使用 Actor 模型，那么该 `PersonRepository` 也应该是一个 actor。在这种情况下，还需要给 `PersonRepository` 定义一个对外可用的接口。因为

actor 是通过消息而不是通过方法进行通信的，所以在这里使用 trait 没有意义。相反，要将接口定义为该域的消息协议，如下所示。

```
object PersonRepository {
  case class Send(userId: UUID, message: Any)
}
```

然后，可以在基础架构中定义一个使用该协议的 PersonRepository，但是如何将该协议用于基础架构呢？以下是实现思路。

```
class CassandraPersonRepository extends Actor {
  ...
}
```

因为 Akka 通过使用 ActorRefs 而不是 actor 实例进行通信的，所以可以在需要的地方传递该仓储的引用。使用该引用的客户端不需要知道它们正在与 CassandraPersonRepository 而不是 SQLPersonRepository 进行通信，它们只需要知道此仓储使用 PersonRepository 协议即可。然后，可以将消息“发送”到仓储，标识该消息所针对的用户。最后由 PersonRepository 找到适当的用户（或创建用户），然后传递消息。

工厂和对象创建

工厂（factory）的主要作用是抽离出创建新的域对象时带来的复杂性。有时，创建新的域对象很复杂，可能涉及将多个部件连接在一起，或从数据存储中提取一些数据。这个过程可能需要执行很多复杂的操作。工厂和仓储的区别其实很小，工厂旨在抽象出新对象的创建，而仓储旨在抽象出对现有对象的重新创建。然而，这种微妙的差异往往不足以创建一个新的抽象层。为此，工厂和仓储有时会被合并使用，提供创建一个新对象或返回一个现有对象（如果可能的话）的方法。

在 Akka 中，工厂的操作与仓储的操作非常相似，不遵循 ask 模式，通常使用 tell 模式来创建对象，然后将消息传递给新创建的实例。由于它与仓储非常相似，所以没必要去严格地区分。

域服务

当使用 DDD 时，我们的目标是尝试将逻辑放入一个现有的域对象中。这通常意味着要将操作添加到聚合根中，但有时这很困难。例如有些操作并不适合加入到任何聚合根中执行，或者相反，有些操作可能和多个聚合根都有联系。在这些情况下，很难找到合适

的域对象来填充所需的角色。对此，我们可以引入一种被称为服务（service）的概念。服务是一个域对象，用于处理不适合作为聚合的操作，但服务可以根据需要与聚合进行交互。

一般来说，我们会把服务作为最后的手段。如果存在一个聚合适合某个角色，应该优先使用聚合。如果现有的聚合都不合适，应该仔细检查是否有疏忽遗漏的其他聚合。只有在用尽所有可能的手段都找不到聚合时，才应该引入服务。

在 Akka 中，服务可以有多种形式。它们可以是长期存在的 actor，和其他聚合 actor 一起工作。或者，它们也可以是专门为执行某个临时任务而创建的临时 actor，在任务完成后就会被终止。

在调度域示例中，服务示例是用来完成特定任务的 worker actor。例如，我们可能希望有一个临时的 worker actor 来处理单个请求。

```
class ScheduleRequestService(request: ScheduleRequest) extends Actor {
  ...
}
```

`ScheduleRequestService` 的任务是管理与该特定请求相关的状态，并且在请求期间与需要的所有聚合进行通信。完成处理请求后，可以终止 `ScheduleRequestService`。另一种实现方法是将 `ScheduleRequestService` 创建为一个长期存在的 actor，这样就不是将请求作为构造函数的参数处理了，而是作为消息接收。但是，为了管理请求的所有状态，可能仍需要创建一个临时 actor（例如 `ScheduleRequestWorker`）。

有界上下文

DDD 的真正关键点不在于用来构建领域系统的那些小的基础构建块，也不是靠聚合或者仓储发挥真正的作用。是因为有界上下文（bounded context），DDD 才会变得如此特殊。任何有一定规模的系统都将自然地分解成更小的组件，这些组件可以有自己的域。虽然这些域可以与整个系统共享一些元素，但是这些元素的表示形式可能会根据使用它们的上下文的不同而有所不同。试图构建一个适合所有用例的单一、内聚的域的想法会很快被证明是行不通的。有界上下文通过识别操作的上下文，可以让不同的上下文具有不同的域模型，进而避免上述问题。

以我们一直都在讨论的调度系统为例。当试图安排从事某项特定工作的人时，有专门用来表示这些人的域，也就是说，有一个子系统负责给这些人安排工作。另一方面，还有另一个单独的子系统将负责维护这些人和关于他们的信息。这两个子系统都处理相同的人，但处理的事情不一样，针对的是人的不同需求。人员管理子系统可能关心如地址和

电话号码之类的信息，而另一个子系统可能不会。如果试图建立一个能实现这两个目的的域模型，那么代码会变得很混乱。为了解决这个问题，所有与工作安排相关的业务功能都应该与人员管理完全无关。当安排工作时，也可能会涉及安排除人之外的其他资源。这些资源可能是硬件、机械或车辆零件。尝试把这方面的业务强行融入人员管理子系统中，一定会失败。

除用于管理人员的子系统之外，系统中可能还存在其他有界上下文。我们需要一个单独的上下文来管理技能，虽然这个上下文最终可能成为人员管理子系统的一部分功能，但是技能管理是非常庞大和复杂的，我们还是希望将其与人员管理分开。另外，项目的细节可能不是实际调度所必需的，在创建项目时，需要包括项目的主要联系人等信息。这些信息虽然很重要，但并不是项目安排需要用到的信息。同样，将项目管理从项目安排中分离出去是有好处的。

在 Akka 中，有界上下文也具有不同的形式。比如可以在系统中创建一系列顶级 actor，每一个 actor 专门用于一个特定的有界上下文。通常，有界上下文表示彼此隔离的服务，它们可能是通过 Akka HTTP 绑定在一起的独立的 actor 系统。或者，它们也可能是通过 Akka Remoting 或 Akka 集群绑定在一起的 actor。它们可能驻留在同一个 Java 虚拟机（JVM）中，但它们也可以位于单独的 JVM、单独的机器，甚至单独的数据中心中。实际上，当把应用程序划分成有界上下文时，会发现虽然一些上下文可以很自然地映射到 Actor 模型中，但另一些可能更适用于使用面向函数的架构或传统的面向对象的架构。关键是要认识到每个有界上下文是独立且不同的，允许根据需求做出不同的决定。我们并没有被束缚于使用某种特定的方法，可以尝试最适合有界上下文的任何方法。

这种有界上下文的方法与现代微服务体系结构非常相似。在这种情况下，每个微服务通常代表一个有界上下文。更重要的是，通过将应用程序分为小的组成模块，可以更好地分发和缩放它们。

这种将应用程序分成单独的有界上下文并在多台机器上分发这些上下文的想法，其实就是分布式领域驱动设计（DDDD）的思想。使用 Akka clustering、集群 sharding 和 Akka HTTP 这样的工具可以将一个大型系统分成单独的有界上下文，然后用 Akka 提供的工具可以很容易地将它们分发到多台机器上。在这里，Actor 模型提供的位置透明性的特性可以自由地在各种不同的模式中分发 actor。无论是通过 Akka HTTP 分发有界上下文，还是使用集群 sharding 在单个有界上下文中分发 actor，都不会被限制该如何部署系统。实际上，部署 actor 的方式只是一个实现细节，而不是应用程序的固有部分，这样可以实现独立缩放系统的不同部分。

结论

总体来说，DDD 可以提供一种方法来创建满足系统需求的结构。在没有这种结构的情况下构建的应用程序往往难以理解和维护，因此整体质量较低。对于使用 Actor 模型构建的 Akka 系统尤其如此，因为期望的高级隔离在没有 DDD 的情况下很难让人看到系统的整体设计。

了解了 Akka 和 Actor 模型（并展示了如何与 DDD 结合）之后，大家便有了构建强大的、可扩展的、高度可维护的系统的必要工具，可以遵循这些已经建立好的模式来保障系统的顺利构建。

在第 4 章中，我们将讨论使用 actor 设计的优秀系统的特性，以及如何更好地组织系统以实现高并发和高效的数据流，同时保持系统的可扩展性。

第 4 章

优秀的Actor设计

了解如何建立良好的 actor 系统一般要从小型结构开始。无论是否使用 Akka，我们在项目中使用大型结构通常是导致软件项目失败的最主要原因。而且如果在理解大结构之前不先理解其内部的小构建块，小构建块中的错误可能会传播到整个系统，从而使代码变得无法维护。这个问题不是 Akka 独有的，如果尝试直接使用函数式编程语言去编写代码而不先了解编程语言的基础知识，也会遇到类似的问题。

在本章中，我们将讨论如何避免这些问题，并应用一些较好的原则和最佳实践来构建 actor 和系统。

大系统小做

正如刚才提到的，理解如何构建良好的 actor 系统需要从小规模的结构开始。我们可以讨论如何构建 actor 结构以及如何把大规模的 actor 连接在一起，但是如果在小规模结构上缺乏良好的设计，仍然很容易遇到瓶颈——阻塞问题的操作和可能导致应用程序失败的混乱的代码。

人们在使用 Akka 时最常犯的错误往往是发生在小规模的构建块上而不是大型结构上。即使这些小问题比架构问题要小很多，但是累积在一起后便会变成不可忽视的大问题。无论是意外暴露可变状态，还是将来可能关闭这些可变状态，都很容易让人犯错，导致 actor 系统变得难以理解。需要记住，只有在正确使用 actor 时，actor 才会帮忙解决构建并发系统时经常遇到的那些问题。如果没有正确使用 actor，则会产生与任何其他系统一样多的并发错误。即使 actor 可以帮助我们解决构建并发系统时的大部分问题，但它们不能解决所有问题。所以和其他并发系统一样，需要注意瓶颈、竞态条件（race condition）和其他可能出现的问题。

回想前面的示例——项目管理系统。在这个系统中，我们分离出了有界上下文，如调度服务子系统、人员管理服务子系统和项目管理服务子系统。但是当部署系统时，我们会遇到一些意料之外的事情。人员管理服务子系统应该是一个非常简单的数据管理服务，却经常意外地丢失数据，没有如预期运行。深入排查，我们发现该系统是混合使用不同的并发范例来构建的。系统里面的 actor 和 future，开发者在完全不了解它们背后意义的情况下直接混合使用，因此产生了竞态条件（race condition）。我们要关闭多线程内的可变状态，避免一系列的并发问题。

与此同时，通过更加深入的排查，我们了解到调度服务不能处理当前的系统负载，因此不得不对它进行扩展。因为它涉及相当复杂的逻辑，而且目前运行得十分糟糕，以至危及了整个项目，因此可以通过扩容来缓解这个问题。但即使扩容也还是会存在运行很慢的问题，进一步调查发现，应用程序只能处理少量的并发任务，因为我们已经引入了会消耗所有线程资源的阻塞型操作，因此削弱了应用程序。

Akka 中有一套模式和原则可以用来防止这些问题，关键点是要知道何时何地去应用这个模式。使用不正确也会引发不少问题，但是如果正确应用它们，它们将为构建系统打下坚实的基础。

封装 actor 中的状态

actor 的一个关键特性是可以以线程安全的方式管理状态，这是一个很重要的功能。另外还有很多管理状态的方法，每种方法都有它的意义和用途，需要根据当前的业务需求去选择合适的方法来管理状态。了解这些模式很有好处，能帮助我们在查看别人的代码时很容易地识别出它们并理解代码的意图。

使用字段封装状态

最简单的状态管理方法是用 actor 中的可变私有字段去记录状态，这是相对简单的实现。如果习惯命令式编程风格，那么一定会很熟悉这种方法。这也是书籍、课程或者在线资料里面提供的比较常见的管理可变状态的方法。

项目调度示例中有“人”的概念。一个人由许多可以被改变的数据字段组成，包括名字、姓氏、业务角色等。这些字段可以单独或作为一个整体被更新。其基本实现如下所示。

```
object Person {
  case class SetFirstName(name: String)
  case class SetLastName(name: String)
  case class AddRole(role: Role)
}
```

```
class Person extends Actor {
  import Person._
  private var firstName: Option[String] = None
  private var lastName: Option[String] = None
  private var roles: Set[Role] = Set.empty

  override def receive:Receive = {
    case SetFirstName(name) => firstName = Some(name)
    case SetLastName(name) => lastName = Some(name)
    case AddRole(role) => roles = roles + role
  }
}
```

习惯于面向对象编程和在 Java 语言中使用 getter 和 setter 的人会比较熟悉这样的格式。如果更倾向于函数式编程思想，那么这种可变状态的存在可能会让人感到沮丧，但 actor 的上下文中的可变状态是安全的，因为它受到 Actor 模型的保护。如果深入到 Scala 的标准库中去看代码，会发现可变状态实际上是相当普遍的。在这种情况下，可变状态被隔离在函数内，因此不会在不同线程之间共享。我们把一个 actor 的可变状态隔离在它内部，而且由于 actor 的单线程模式，它的可变状态也不会被并发访问。

当尝试对一些简单的东西建模时，用 actor 的可变私有字段去记录状态是一种不错的选择。例如在涉及的字段和值比较少，相应的逻辑处理代码也不是很多的情况下，这种方法会是比较好的候选方案。

这种方法的问题在于，随着复杂性的增加，该方法会变得难以管理。在用 actor 代表一个人的例子中，若我们想对它进行进一步扩展，需要保持各种附加信息（如地址，电话号码等）的状态时，会发生什么？我们正在向 actor 添加更多字段，而且所有字段都是可变的，虽然它们是安全的，但随着时间的推移，这种字段的增长会使系统代码变得丑陋。如果和这些字段相关的逻辑代码本身也是比较复杂的，又会如何？——比如设置人员地址的时候需要有验证地址有效性的逻辑代码，这会使 actor 变得越来越臃肿，而且添加越多，就会变得越糟，如下所示。

```
class Person extends Actor {
  private var firstName: Option[String] = None
  private var lastName: Option[String] = None
  private var address: Option[String] = None
  private var phoneNumber: Option[String] = None
  private var roles: Set[Role] = Set.empty

  override def receive:Receive = {
    case SetFirstName(name) =>
```

```
        firstName = Some(name)
      case SetLastName(name) =>
        lastName = Some(name)
      case SetAddress(address) =>
        validateAddress(address)
        address = Some(address)
      case SetPhoneNumber(phoneNumber) =>
        validatePhoneNumber(phoneNumber)
        phoneNumber = Some(phoneNumber)
      case AddRole(role) => roles = roles + role
    }
  }
```

另一个问题是 actor 本身会比 Scala 更难测试，因为它们是并发的，并且这种并发会将复杂性引入到测试中。随着代码的复杂性增加，测试的复杂性也会增加。很快，我们可能会发现自己的测试代码变得难以理解且难以维护，这又是一个问题。

解决这个问题的方法是把相关的逻辑代码从 actor 里面抽取出来。可以创建能够堆叠到 actor 上的 trait，这些 trait 包含改变可变状态的逻辑代码。然后该 actor 就变成只包含与并发机制相关的逻辑代码了。所有改变状态的逻辑都已经被提取到辅助 trait 里面了，它们可以作为纯函数被编写和测试。在 actor 中，只需要担心可变性和并发方面的问题，无须担心其他问题。代码如下。

```
trait PhoneNumberValidation {
    def validatePhoneNumber(phoneNumber: Option[String]) = {
      ...
    }
}

trait AddressValidation {
    def validateAddress(address: Option[String]) = {
      ...
    }
}

class Person extends Actor with PhoneNumberValidation with AddressValidation {
...
}
```

这种方法可以简化 actor，也减少了并发测试的复杂度。然而它并没有减少可变字段的数量，针对这些字段需要做的测试也没有减少。但这仍不失为一个不错的手段，可以防止 actor 变得太大而难以管理。

那么如何才能减少可变状态的数量呢？怎么将一个 actor 的可变状态变少呢？有几个可

行的方案。一种方法是重构 actor 的结构。也许我们实际上并不想得到一个包含所有字段的 actor。在某些情况下，将此 actor 分为多个 actor，每个 actor 管理一个字段或一组字段可能会更好。是否应该这样做，很大程度上取决于当前域的情况，这不是一个技术问题，而是一个领域驱动设计（DDD）问题。需要查看当前域的情况，再决定是把用到的字段组合到一起处理还是分别处理，这些字段在逻辑上是域中单个实体的一部分还是作为单独实体存在，这将有助于决定是否需要分离出额外的 actor。

但是如果确定它们在逻辑上是同一个实体的一部分，因此判定为同一个 actor，这种情况该如何处理？是否有可能在单个 actor 中减少一些可变的状态呢？

使用“状态”容器封装状态

在前面的示例中，我们可以将一些 Account 的逻辑提取到单独的基于非 actor 的 `PersonalInformation` 类中。所以如果将这些逻辑提取成一个类或一组类，它们会更安全，更容易测试，并能处理更多的事务。只需要有一个 `var` 变量来存储整个状态即可，而不需要有多个。代码如下。

```
object Person {
  case class PersonalInformation(
    firstName: Option[FirstName] = None,
    lastName: Option[LastName] = None,
    address: Option[Address] = None,
    phoneNumber: Option[PhoneNumber] = None,
    roles: Set[Role] = Set.empty
  )
}

class Person extends Actor {
  private var personalInformation = PersonalInformation()

  override def receive: Receive = ...
}
```

这是一个改进。现在有一个更容易测试的 `PersonalInformation` 类，它可以将所需的所有验证逻辑放入该状态对象，并根据需要调用里面的方法。可以以不变的方式编写逻辑代码，并且像纯函数那样去进行测试。事实上，如果决定以后不再使用 actor，完全可以消除 actor 而不必过多改变 `PersonalInformation` 中的内容（或者不做任何改变）。甚至可以选择不在 actor 的伴生对象中声明 `PersonalInformation`，而是将该逻辑移动到同一个包中的另一个位置，或将其移动到其他包或模块中。

当模型的逻辑变得复杂时，上述方法是一个很好的技术手段。它提供了提取所有业务逻

辑的方法，以便将这些逻辑封装在 PersonalInformation 对象中，这样 actor 就变成了纯基础结构。这样一来，受单线程模式保护的 var 变量便可以把内部的可变状态以及相关逻辑消除得差不多了。但是如果想完全消除，能做到吗？

使用 become 封装状态

状态管理的另一种方法是使用 become 功能。这种特殊的技术也更符合前面讨论的想法：行为和状态的变化事实上是一回事。使用这种技术可以完全消除 var 变量，同时不牺牲性能。代码如下。

```
object Person {
  case class PersonalInformation(
    firstName: Option[FirstName] = None,
    lastName: Option[LastName] = None,
    address: Option[Address] = None,
    phoneNumber: Option[PhoneNumber] = None,
    roles: Set[Role] = Set.empty
  )
}

class Person extends Actor {

  override def receive: Receive = updated(PersonalInformation())

  private def updated(personalInformation: PersonalInformation):Receive = {
    case SetFirstName(firstName: FirstName) =>
      context.become(updated(personalInformation
        .copy(firstName = Some(firstName))))
    ...
  }
}
```

这种方法完全消除了对 var 变量的需求。现在已经不是通过操纵 var 来改变状态，而是要通过改变行为来改变状态。通过更改行为来让下一个消息具有与上一个消息不同的 PersonalInformation——这看上去很好，因为更有“面向函数”的感觉（没有更多的 var 变量），同时也更好地映射到 Actor 模型中。记住，在 Actor 模型中，状态和行为是一回事，在代码中可以反映出来。

需要警惕的一点是，虽然这种技术在有些情况下是相当有用的，特别是当建立有限状态机时，但是它的代码会更复杂，而且需要花更多的时间来理解。这种技术不像操纵 var 变量一样简单，如果要调试它，会发现跟踪代码变得异常困难，特别是在牵扯其他行为转换的情况下。

所以，什么时候应该使用这种方法而不使用 var 变量呢？对于具有多个行为的情况，更确切地说，这些行为可以包含不同的状态对象时，使用这种技术才最合适。稍微修改一下这个例子。在该示例中，大部分数值都被设置为 Options，这是因为在创建 Person 时，这些值可能尚未初始化，稍后才会被初始化。所以，需要一种方法来应对这种情况。但是，如果在分析域的时候意识到一个人总是用当前已存在的信息被创造出来，那么该如何利用 actor 来应对这种情况呢？这里提供一种方法，代码如下。

```
object Person {
  case class PersonalInformation(firstName: FirstName,
                                 lastName: LastName,
                                 address: Address,
                                 phoneNumber: PhoneNumber,
                                 roles: Set[Role] = Set.empty
                                 )

  case class Create(personalInformation: PersonalInformation)
}

class Person extends Actor {

  override def receive: Receive = {
    case Create(personalInformation) =>
      context.become(created(personalInformation))
  }

  private def created(personalInformation: PersonalInformation): Receive = {
      ...
  }
}
```

这个例子中有两个新的状态，分别是初始状态和创建状态。初始状态下尚未提供个人信息，因此，这时 actor 只接受一个 create 的命令。转换到创建状态后，可以处理其他消息，从而允许设置那些单独的字段。这种方法消除了对可选（optional）值的需求，因为 actor 只会以两种状态（未填充的或完全填充的）中的一种存在。在这种情况下，状态对象仅在特定状态下有效，这个例子中是“创建”状态。可以使用 var 变量并将其设置为相应的选项，但是需要不断检查该选项的值，以确保它是期望的值。通过使用 become 捕获状态，确保只处理在当前状态下有效的消息类型，这也确保了状态总是我们期望的，而不需要额外检查。

当存在多个行为转换，且其中每个转换又可能具有不同的状态时，最好通过使用 become 去捕获该状态，这样可以简化逻辑，而不使其更复杂，同时降低了认知开销。

通过组合使用 become、可变字段和功能状态对象，可以获得准确反应应用程序域的 actor。不可变状态对象使我们可以创建丰富的域结构，独立于 actor 进行完全测试。我们可以使用 become 来确保 actor 只能在一组有效状态中存在，不需要为不重要的操作创建默认值。当 actor 足够简单时，可以使用更便捷更常见的构造（例如用具有不可变集合的可变字段）来编码实现它们。

将 future 与 actor 混合

future 是将并发引入系统的有效途径，但是当把它们与 Akka 组合使用时，需要特别小心。因为 future 提供了一种并发模型，然而 Akka 却提供了另一种不同的模型。每个模型都有自己坚持的设计原则，这些原则往往不能很好地一起工作。应该尽量避免在单个上下文中混合使用不同的并发模型，但有时却不得不混合使用它们。对于这种情况，需要特别遵循那些可以确保我们安全地混合使用 future 和 actor 的设计模式。

问题的核心在于，我们正在组合两种截然不同的并发模型。future 和 actor 在并发的处理方式上有很大的不同，future 会破坏 actor 提供的单线程模式。来看一个可以破坏单线程模式的简单例子，代码如下。

```
trait AvailabilityCalendarRepository {
  def find(resourceId: Resource): Future[AvailabilityCalendar]
}
```

这是一个非常简单的使用 future 的方法。在这个例子中，仓储通过使用 future 来访问数据库。我们知道，访问数据库与其他操作相比需要花费更多的时间，而且可能访问失败。

使用 future 本身不存在任何问题，但是当在某一个 actor 的上下文中使用仓储的时候，问题就会出现了。如果要把这个仓储结合到 actor 系统中使用，则需要先创建一个所谓的 Actor 封装器，最初的实现代码如下。

```
object AvailabilityCalendarWrapper {
  case class Find(resourceId: Resource)
}

class AvailabilityCalendarWrapper(calendarRepository:
   AvailabilityCalendarRepository) extends Actor {
  import AvailabilityCalendarWrapper._

  override def receive: Receive = {
    case Find(resourceId) =>
      calendarRepository
        .find(resourceId)
        .foreach(result => sender() ! result)
```

```
        // WRONG! The sender may have changed!
    }
  }
```

这似乎相当简单。代码接收到一个 Find 消息，提取 ResourceId 并调用仓储，然后将结果返回给发送方。这里的问题在于，它打破了单线程模式。这段代码中的 foreach 是在 future 机制上运行的。这意味着，运行此代码时可能在不同的线程中工作，actor 已经继续处理后续的任务了，在 foreach 运行的时候，actor 可能正在处理另一个消息。这种情况下，发送方可能已经改变了，不再是原来的那个了。

有多种方法可以解决上述问题。可以创建临时值以保存正确发送方的引用，代码如下。

```
case Find(resourceId) =>
  val replyTo = sender()
  calendarRepository
    .find(resourceId)
    .foreach(result => replyTo ! result)
```

理论上这样就可以解决问题。ReplyTo 将被设置成正确的 sender() 的值，之后也不会被改变。但是由于一些原因，这并不是最理想的解决方案。比如，如果我们并不只是需要引用 sender 这一个值，或者还有其他可变状态需要操作，该怎么办？这种情况该如何处理？前面的代码还有一个根本的问题：如果不看仓储的签名代码，该怎么验证我们正在使用 future？该 foreach 操作可以很容易在一个选项或集合上进行。虽然可以切换为使用 Scala 的 for 语法或更改为使用 onComplete 回调，但仍然没有突出显示此操作已成为多线程的这一事实。

更好的解决方案是在 Akka 中使用管道（pipe）模式。使用管道模式就可以使用 future 了，然后将 future 的结果“用管道输送”到另一个 actor 中。结果是，那个 actor 会在某个时刻接收到 future 返回的结果或接收到 Status.Failure（如果 future 失败的话）。更改之前的代码，使用管道模式，如下所示。

```
import akka.pattern.pipe

object AvailabilityCalendarWrapper {
  case class Find(resourceId: Resource)
}

class AvailabilityCalendarWrapper(calendarRepository:
   AvailabilityCalendarRepository) extends Actor {
  import AvailabilityCalendarWrapper._

  override def receive: Receive = {
```

```
      case Find(resourceId) => calendarRepository.find(resourceId).pipeTo(sender())
    }
  }
```

使用管道模式有很多优点。首先可以维持单线程模式，因为 future 的结果是以另外一种消息方式被返回给 actor 的，而且接收到这个结果的时候并不需要以任何并发方式去访问该 actor 的状态。本示例通过调用 PipeTo 解析 sender，而且在 future 完成工作之前并不会向 sender 发送消息。所以即使 actor 继续处理后续的消息，对应的 sender 仍然可以一直保持之前的正确状态。

管道模式的另一个好处是，它是显式的。当看到它的时候，并不会产生“代码是否涉及并发运行”的疑问。因为 pipeTo 这段代码很明显地显示，操作并不会立即发生，而是会发生在将来的某个时间点。这就是管道模式，不需要去查看仓储的签名代码就能知道里面的逻辑涉及 future 的相关操作。

进一步探讨一下这个问题。如果不想立即将结果发送给 sender 怎么办？如果想先对其执行一些其他操作（如修改数据使其使用不同的格式）该怎么办？这会对上面的示例产生什么样的影响？最简单的实现如下。

```
import akka.pattern.pipe

object AvailabilityCalendarWrapper {
  case class Find(resourceId: Resource)
  case class ModifiedCalendar(...)
}

class AvailabilityCalendarWrapper(calendarRepository:
   AvailabilityCalendarRepository) extends Actor {
  import AvailabilityCalendarWrapper._

  private def modifyResults(calendar: AvailabilityCalendar):
     ModifiedCalendar = {
    // Perform various modifications on the calendar
    ModifiedCalendar(...)
  }

  override def receive: Receive = {
    case Find(resourceId) =>
      calendarRepository
        .find(resourceId)
        .map(result => modifyResults)
        .pipeTo(sender())
  }
```

```
}
```

像以前一样，乍看之下不错。这段代码仍然使用管道模式，所以 sender 还是安全的。但是代码在 future 上引入了 .map 操作，在这个 .map 中调用了一个函数。现在的问题是，.map 正在单独的线程中进行操作，这也创造了打破单线程模式的机会。一开始代码可能是完全安全的，但是随后有人可能会在 modifyResults 中访问 sender，或者在 calendar 中访问其他可变状态，因为它在 AvailabilityCalendar Wrapper 中的一个函数内部，所以我们可能会假定访问该可变状态是安全的。这样的函数并不能很明显地被看出来是在 future 上调用的，所以很容易在无意识的情况下更改在单独上下文中执行的代码，从而引发问题。那么该怎么解决呢？此处提供一种方法，代码如下。

```
import akka.pattern.pipe

object AvailabilityCalendarWrapper {
  case class Find(resourceId: Resource)
  case class ModifiedCalendar(...)
}

class AvailabilityCalendarWrapper(calendarRepository:
   AvailabilityCalendarRepository) extends Actor {
  import AvailabilityCalendarWrapper._

  override def receive: Receive = {
    case Find(resourceId) =>
      calendarRepository.find(resourceId).pipeTo(self)(sender())
    case AvailabilityCalendar(...) =>
      // Perform various modifications on the calendar
      sender() ! ModifiedCalendar(...)
  }
}
```

这段代码删除了 .map 操作，在 future 解析之后用回了 pipeTo 操作。不过这一次，pipe 返回了 self，而且将 sender 作为 pipeTo 的第二个参数使用，允许维持对原始 sender 的引用。这样可以明确区分出哪些操作是将来才会执行的，哪些操作是现在立即就会执行的，就不可能会破坏单线程模式了。

有时候我们会发现 future 返回的结果很一般，都是像整数或字符串这样的返回。在这种情况下，简单地使用管道模式会使代码变得混乱，因为 actor 将接收一个描述性不强的简单类型。如果可以使用某种方式丰富这些类型会更好，例如将其包装在一条消息中。在这种情况下，为了将消息转换为适当的包装器，使用 .map 是合适的，只要保持操作尽可能简单和独立即可。代码如下。

```
ourFuture.map(result => Wrapper(result)).pipeTo(self)
```

我们使用 .map 进行的唯一操作就是构造包装器类型。在 future 上使用这种类型的 .map 被认为是低风险的，尽管这个代码也引入了一些风险，但可以在一个 actor 内部使用。Wrapper 在哪里定义？它在构建期间会在某处访问可变状态吗？在编写代码的时候有可能会犯一些错误，但只要遵循最佳实践，错误就可以避免。也是出于这个原因，这样的代码通常被认为是可以接受的。

需要记住，在一个 actor 中有许多状态可以被认为是可变的状态。“sender”当然是可变状态的一种；可变 var 变量或可变集合是可变状态；actor 的上下文对象也包含可变状态，例如 Context.become 是在使用可变状态；在 receive 方法的参数中声明的一个不可变的 val 变量可能也是可变的状态，因为 receive 方法可以改变；甚至对支持锁的数据库的访问仍然可能构成可变状态，虽然它看上去是“安全的”，但实际上它改变了系统的某些状态。

一般来说，应该尽可能地在 actor 中使用纯函数。这些纯函数永远不会对 actor 的任何状态进行读写操作，而是依赖于传入的参数对其操作，并返回修改过的值，然后使用它们安全地修改 actor 中的可变状态。纯函数也可以从 actor 中被提取出来，这样可以确保我们不会在 actor 内部访问任何可变状态，从而可以在一个 actor 的上下文中以线程安全的方式使用这些函数。

当在一个 actor 中使用非纯函数时，需要确保在单线程模式的上下文中执行操作。为了做到这一点，当和 future 一起工作时应该选择管道模式。有时，创建一个包装器可能是更好的选择，包装器可以把 future 隔离出来，所以只需要在某个环节多加注意，在系统的其余地方则可以完全忽略它。更好的方案是，完全避免使用 future，而使用其他技术替代。

Ask 模式和替代方案

Ask 模式是 Akka 中的一种常见模式。它的用法非常简单，提供了非常具体的用例。简单介绍一下它的工作原理。

假设想使用上文介绍过的被“包装过”的actor，发送“find”消息给它，然后需要它返回结果。若使用 Ask 模式执行此操作，代码如下。

```
import akka.pattern.ask
import akka.actor.Timeout
import scala.concurrent.duration._
```

```
implicit val timeout = Timeout(5.seconds)
val resultFuture = (availabilityCalendar ?
   AvailabilityCalendarWrapper.find(resourceId)).mapTo[AvailabilityCalendar]
```

在此示例中，我们使用 Ask 模式来获取 `Future[Any]`。然后使用 `mapTo` 函数将 future 转换为 `Future[AvailabilityCalendar]`。

若希望某个操作在一定的时间内成功执行并返回，使用 Ask 模式会非常有效。发生故障时将收到失败的 future，如果操作过长或不完整，也将收到失败的 future，对于有用户在另一端等待结果的情况，该模式非常有用。因此，当构建 Web 应用程序、REST API、其他具有有效时间限制的应用程序时，使用 Ask 模式是相当常见的。

Ask 模式的问题

使用 Ask 模式需要小心。它本身没有什么问题，在许多情况下，它就是最符合我们预期的方法，但是过度使用 Ask 模式会出现问题。Ask 模式给人一种很熟悉的感觉，它与函数式编程的工作方式相似：用一组参数调用一个函数，该函数返回一个值。此处不是一个函数，而是一个 actor。但是，函数式模型是强调请求 / 响应模式的，而 Actor 模型通常更合适使用“Tell，Do Not Ask”的工作方式。

下面举一个例子来说明过度使用 Ask 模式可能会引发的问题。为了专注于问题，本例将脱离业务域。

假设有一系列的 actor，每个 actor 都需要执行一些小任务，然后将结果传递给管道中的下一个 actor。最后，原始 actor 接收最终结果。假设整个操作需要在 5 秒的时间窗口内发生。来看以下代码。

```
import akka.pattern.{ask, pipe}

case class ProcessData(data: String)
case class DataProcessed(data: String)

class Stage1() extends Actor {

  val nextStage = context.actorOf(Props(new Stage2()))

  override def receive: Receive = {
    case ProcessData(data) =>
      val processedData = processData(data)
      implicit val timeout = Timeout(5.seconds)
      (nextStage ? ProcessData(processedData)).pipeTo(sender())
  }
```

```
  }

  class Stage2 extends Actor {

    val nextStage = context.actorOf(Props(new Stage3()))

    override def receive: Receive = {
      case ProcessData(data) =>
        val processedData = processData(data)
        implicit val timeout = Timeout(5.seconds)
        (nextStage ? ProcessData(processedData)).pipeTo(sender())
    }
  }

  class Stage3 extends Actor {

    override def receive: Receive = {
      case ProcessData(data) =>
        val processedData = processData(data)
        sender ! DataProcessed(processedData)
    }
  }
```

这段代码会正常工作并且会完成工作。但在解决方案中存在一些奇怪的地方。第一个问题是超时设定，每个阶段（除了最后一个）都有一个 5 秒的超时设定。这是一个问题，假设第一个阶段消耗了 5 秒，第二个阶段消耗的时间不到 5 秒（比如 1 秒），所以下一个阶段可能不需要 5 秒的超时时间，只需要 4 秒就够了。第三个阶段只用了 4.5 秒的时间，所以第二个阶段不会超时，而将成功执行任务。但是整个执行过程还是失败了，因为第一个阶段超过了 5 秒的时间限制。这里的问题是，第一个发生的超时（5 秒限制）对于整个应用程序很关键。这个需要在 5 秒时间窗口内成功执行的操作非常关键，因为它的超时会导致之后的超时都变得无关紧要，无论它们需要执行 10 秒还是 2 秒都没有关系，只有第一个超时才对系统有影响。因为过度使用 Ask 模式，我们被迫在过程的每个阶段引入超时机制，这些超时时间使代码变得混乱。最好是可以修改解决方案，使得只有第一个超时是必要的。

这里的关键在于，每次在系统中引入超时机制时都需要考虑是否需要，这样有意义吗？如果超时了，是否可以采取一些操作来纠正问题？如果发生故障，需要通知某人吗？如果这些问题的回答是“是”，则 Ask 模式可能是正确的解决方案。但是，如果已经有类似超时的机制可以处理这种情况，那么应该尽量避免使用 Ask 模式，而使用现有的机制。

附带的复杂性

上一个示例中的代码还有另一个问题：使用 Ask 模式使系统变得更加复杂了。系统创建了临时的 actor 专门负责等待返回的结果。虽然临时 actor 并不十分消耗资源，但终究不是全无负担，多少还是要消耗一些系统资源。图 4-1 为过度使用 Ask 模式的示意图。

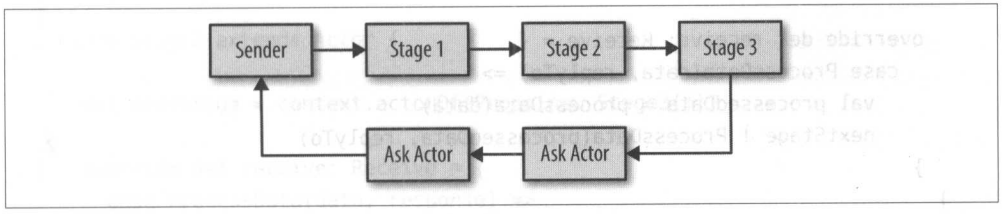


图4-1　Ask的过度使用

从上图中可以看到，看似简单的 Ask 模式中包含许多细节。

我们创建了一个比真正需要更复杂的消息流。引入了至少两个必须由系统维护的临时 actor。这些都会消耗系统资源。如果看图表并尝试推理，会发现所有额外的复杂逻辑都不是我们真正想要的。我们想要的是绕过中间 actor 将响应直接发给发送者，这是 Tell，而不是 Ask。

Ask 的替代方案

有多种方法可以修复管道问题。其中一个是使用 `forward` 方法，通过转发消息而不是使用 Tell 运算符，确保管道中的最后一个阶段还拥有原始发送方的引用。这种方法消除了超时机制的需要，它将消除作为中间人的临时 actor，代码如下。

```
class Stage1() extends Actor {

  val nextStage = context.actorOf(Props(new Stage2()))

  override def receive: Receive = {
    case ProcessData(data) =>
      val processedData = processData(data)
      nextStage.forward(ProcessData(processedData))
  }
}
```

消除 Ask 模式的另一种方法是将 `replyTo` actor 作为消息的一部分进行传递，即让消息包含那些需要被回应的 actor 的引用。这样在管道的任何阶段，我们都可以访问该 actor 并向其发送结果，或者在特定情况下提早退出管道，代码如下。

```
case class ProcessData(data: String, replyTo: ActorRef)
case class DataProcessed(data: String)

class Stage1() extends Actor {

  val nextStage = context.actorOf(Props(new Stage2()))

  override def receive: Receive = {
    case ProcessData(data, replyTo) =>
      val processedData = processData(data)
      nextStage ! ProcessData(processedData, replyTo)
  }
}

class Stage2 extends Actor {

   val nextStage = context.actorOf(Props(new Stage3()))

   override def receive: Receive = {
     case ProcessData(data, replyTo) =>
       val processedData = processData(data)
       nextStage ! ProcessData(processedData, replyTo)
  }
}

class Stage3 extends Actor {

  override def receive: Receive = {
    case ProcessData(data, replyTo) =>
      val processedData = processData(data)
      replyTo ! DataProcessed(processedData)
  }
}
```

最后一个办法是将 Promise 作为消息的一部分来传递。可以通过管道发送 Promise，以便管道的最后阶段可以完成该 Promise。然后，链条的顶层 actor 将可以访问 Promise 正在处理的 future，这时候可以结合 pipeTo 来正确处理 future。操作代码如下。

```
case class ProcessData(data: String, response: Promise[String])

class Stage1() extends Actor {

  val nextStage = context.actorOf(Props(new Stage2()))

  override def receive: Receive = {
```

```
    case ProcessData(data, response) =>
      val processedData = processData(data)
      implicit val timeout = Timeout(5.seconds)
      nextStage ! ProcessData(processedData, response)
  }
}

class Stage2 extends Actor {

  val nextStage = context.actorOf(Props(new Stage3()))

  override def receive: Receive = {
    case ProcessData(data, response) =>
      val processedData = processData(data)
      implicit val timeout = Timeout(5.seconds)
      nextStage ! ProcessData(processedData, response)
  }
}

class Stage3 extends Actor {

  override def receive: Receive = {
    case ProcessData(data, response) =>
      val processedData = processData(data)
      response.complete(Success(processedData))
  }
}
```

这些方法中的每一种都是有效的，根据实际情况，可能其中某种方法会比其他方法更合适。关键是要意识到，我们应该在非常特殊的情况下使用 Ask 模式。当需要与系统外部的 actor 通信，或者需要使用带有有效超时时限的请求 / 响应方法时，Ask 模式是可取的。如果在当前业务并不需要的情况下引入一个超时机制，那很可能是用错了地方。

命令与事件

和 actor 相关的消息可以分为两大类：命令（commands）和事件（events）。

一条命令是指对将来某一个时刻发生的事情的请求，当然最后可能会发生也可能不会发生。例如，如果该命令消息违反了 actor 的某些业务规则，则它可能会选择拒绝或忽略该命令。

事件是记录已经发生的动作的消息。它是过去的事情，不能被改变——系统的其他 actor 或成员可以对它做出反应，但是不能修改它。

将 actor 的协议(此 actor 收到和发出的消息的类和对象)分解为命令和事件是很有帮助的，这样 actor 收到和发出的内容会变得很清晰。请记住，一个 actor 可以接收命令和事件消息，并且同时向外界发出两种类型的消息。通常可以将一个 actor 的消息协议作为它的 API，其实这正是定义 actor 的输入和输出的过程，输入通常定义为命令，而输出通常定义为事件。示例如下。

```
object ProjectScheduler {
  case class ScheduleProject(project: Project)
  case class ProjectScheduled(project: Project)
}
```

在以上代码中，我们为 `ProjectScheduler` actor 定义了一个非常简单的消息协议。`ScheduleProject` 本身是一条命令消息，我们正在指示 `ProjectScheduler` 执行一个操作，只要该操作尚未完成，就有可能会失败。另一方面，`ProjectScheduled` 是一个事件，代表了过去已经发生的事情。因为已经发生，所以不可能再失败。所以 `ScheduleProject` 是 actor 的输入，`ProjectScheduled` 是 actor 的输出。如果不使用 actor，又想通过函数表现出来，则代码如下。

```
class ProjectScheduler {
  def execute(command: ScheduleProject): ProjectScheduled = ...
}
```

了解系统中的命令消息和事件消息之间的区别是非常重要的，它可以帮助消除系统模块之间的依赖关系。考虑以下常见问题：为什么 `ProjectScheduled` 是 `ProjectScheduler` 消息协议的一部分，而不是作为接收它的 actor 协议的一部分呢？简单的答案是，它是一个事件而不是命令。更完整的答案是，这样避免了双向依赖。如果我们将其作为接收方 actor 协议的一部分，发送方将需要知道 `ProjectScheduler` 的消息协议，`ProjectScheduler` 也需要知道发送方的消息协议。通过将命令和生成的事件保持在相同的消息协议中，`ProjectScheduler` 无须知道发送方的任何信息，这有助于使应用程序解耦。

正如刚才所说的，一个常见的做法是将 actor 的消息协议放入其伴生对象中。有时会希望在多个 actor 之间共享消息协议，这种情况下可以创建协议对象来包含消息，代码如下所示。

```
object ProjectSchedulerProtocol {
  case class ScheduleProject(project: Project)
  case class ProjectScheduled(project: Project)
}
```

构造函数的依赖注入

当为 actor 创建 Props 时，可以包含对另一个 actor 的显式引用。这很简单，而且往往是一个很好的方法。

以下是创建 actor 时直接注入另一个 `ActorRef` 的代码。

```
val peopleActor: ActorRef = ...
val projectsActor: ActorRef = system.actorOf(ProjectsActor.props(peopleActor))
```

但是，如果 actor 有多个引用，或因为实例化的顺序问题导致在接收方 actor 构建之前必要的引用不可用，那便应该使用其他方法。

使用路径查找 actor

解决上述问题的一个方法是通过其路径找到想要的 actor，而不是引用。这要求我们必须知道 actor 的路径，当然也可以通过目标 actor 的伴生对象中的静态值访问它。

路径只能解析为 `actorSelection`，而不是实际的 actor 引用。如果想要访问 actor 的引用，必须发送标识信息，并通过返回的回应消息得到引用。或者说，只需要简单地将消息直接发送给 actor selection 即可。

actor selection 有其自身的问题。一个 actor selection 只有当接收到 `Identify` 消息后才会被验证，这意味着另一端可能是一个 actor，也可能不是。反过来意味着，当向 actor selection 发送消息时，不能保证另一端有一个 actor 接收它。此外，actor selection 可以包含通配符，这意味着另一端可能有一个以上的 actor，发送消息时，实际上是将其广播给所有 actor。这可能不是我们想要的效果。

如果不小心使用 actor selection，也可能导致更多的依赖。通常情况下，使用 actor selection 表示我们并没有仔细考虑过 actor 系统的层次结构。使用 actor selection 是因为需要引用另外一个层次系统里的 actor，这会在 actor 系统的不同区域之间产生耦合，并使系统更难以划分或分发。可以将 actor selection 视为类似于使用单例对象乱写的代码，但这通常被认为是一个不好的做法。

将 actor 引用作为一个消息

改变 actor 之间复杂依赖关系的一个更好的方法是将 actor 引用作为一个消息发送给需要它的 actor——例如，向给我们发消息的 actor“介绍自己”的有效方法是，将自己的引用通过特殊的类型封装到消息里面（这样对方接收消息的方法在接收到消息后会很容易区分出来），然后目标 actor 可以保存该引用供以后使用。

Ask 模式是在消息协议中使用 replyTo actor 时被提出的。replyTo 是将一个 actor 引入另一个 actor 的方法，以便第一个 actor 知道发送消息的位置。

使用这种方法时需要思考更多，我们要设计层次结构，使得相关的 actor 可以彼此通信。这种额外的思考是很有益处的，这意味着我们必须考虑应用程序的整体结构，不能仅仅考虑单个 actor。但是也要非常谨慎——如果发现通过消息来传递 actor 的引用须花费太多的精力，很有可能是因为层次结构设计得不合理。

结论

总之，通过掌握一些基本的原则和技巧，我们可以创建保持高内聚的 actor，并且避免不必要的阻塞和复杂性，同时允许 actor 系统中的数据流尽可能平滑和并发地流动。

现在大家已经知道单个 actor 如何能被更好地构造出来，后面我们将提高抽象级别，考虑整个系统中的数据流，以及 actor 正确交互的方式，进而支持这些数据流。

第 5 章

数据流

在第 4 章中，我们介绍了一些实现单个 actor 的方式，以及在执行此操作时遇到的一些陷阱。现在，我们需要考虑如何使用这些 actor 来构建系统。一个单独的 actor 并没有太大的用处，只有将它们结合使用才能发挥 Akka 的真正潜力。本章将介绍使用多个 actor 时要遵循的规范，还会介绍 Akka Streams 的概念，它为流消息提供了一个领域专用术语。

吞吐量与延迟

在讨论如何有效地结合 actor 之前，先来讨论吞吐量与延迟。

当我们测量代码以确定其性能指标时，通常会测量特定操作需要执行的时间。从该操作开始的那一刻起直到结束为止。这个过程可能需要 10 毫秒或 10 秒，这个时间间隔称为延迟。

有时，我们会测量特定时间内可以处理的操作数，而不是处理特定操作需要的时间。我们不关心每个操作需要多长时间，关心的是可以在固定时间内完成多少次操作。例如，我们可以测量 10 秒内执行多少次操作，这个指标称为吞吐量。

无论是延迟还是吞吐量，都需要根据当前领域的业务情况去考虑。在某些领域中，10 秒的延迟甚至 10 小时的延迟都可能是合适的，而在其他领域，这个时间可能太长。需要根据不同的业务场景去设置相应的值，判断这些值是否满足当前的需求，能否提供好的用户体验。

但是当构建高度并发系统时，不能简单地只看某一个性能指标，还要同时考虑吞吐量和延迟。如果只测量一个，则会因为没有了解整个系统的情况导致对系统误判。来看几个具体的例子。

假设用户登录到系统，测量出登录操作的延迟时间大概为 500 毫秒。从用户的角度来看，500 毫秒其实是很快的，几乎察觉不到。如果登录真的只需要 500 毫秒的话，那么用户可能会很高兴，系统可以算作是工作正常。但这只是测量的值，当实际打开系统时，会发现用户需要等待长达 10 秒的时间才能登录！发生了什么？

因为我们只测量了延迟，没有考虑到（无论什么原因）系统其实是以单线程模式在运行的。也许所有操作都被某一个共享资源阻塞了，也许只分配了一个线程去处理这些操作。无论哪种情况，如果每个用户登录都需要 500 毫秒，并且有 20 个用户同时尝试登录，那么这些用户将要排队等待，否则它们将登陆失败。所以，虽然第一个用户能够在 500 毫秒内登录成功，但队列中的最后一个人不得不等待 10 秒。

如果只测量吞吐量呢？这样做时，会发现 500 位用户都能在 10 秒内登录。在这种情况下，在 10 秒内登录人数是 500 似乎是合理的。所以就不需要关注其他的事情了吗？

在这种情况下，我们测量了吞吐量，但忽略了延迟。系统可以在 10 秒内允许 500 位用户登录，但是并没有发觉每个用户需要等待的时间都很长。如果每个用户需要 10 秒的时间完成登录，那么这样的用户体验依然很糟糕。如果应用程序持续需要 10 秒才能登录，那么用户很有可能都会离开。

只有同时测量吞吐量和延迟才能真正了解系统的整体情况。不仅需要了解一个操作需要耗费多长时间，而且还要了解一次可以同时完成多少操作。使用这些信息可以检测到系统中真正的瓶颈，并努力解决它们。

我们已经了解了吞吐量和延迟的基础知识，下面来看一看如何构建在两个指标上都被优化的 actor 系统。

流

当构建软件系统时，我们经常谈论构建数据流（streams）。数据流是指数据经过一系列步骤处理，一条接一条地产生输出的过程。一般来说，当我们谈论流时，会有一个对顺序的假设。如果将两条数据放入流中，则会希望其输出结果以输入的顺序返回。而引入并发机制时就不一样了，按顺序发送的两个输入所产生的输出结果的顺序会发生改变。有时没关系，但有时会有影响。那么，如何使用像 Akka 这样的并发系统来构建系统，同时可以保证流的顺序呢？

回想一下 Akka 中的 actor 使用的邮箱系统。这个邮箱系统在行为上很像先进先出的队列，可以保证无论以什么顺序输入到一个 actor 内，输出都将按预期顺序返回。正是这样一个有序邮箱的概念才使我们可以使用 Akka 来构建流。

可以使用这样的有序邮箱机制在一组离散的 actor 之间传递信息。如果非常谨慎地进行操作，就可以在保持并发性的同时保证顺序。图 5-1 显示了这个过程。

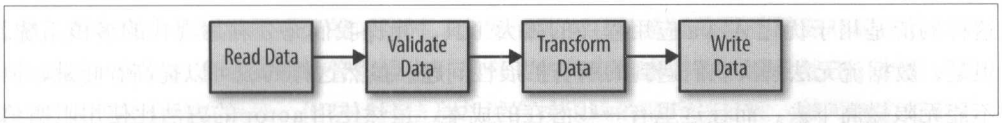


图5-1　actor用于维护消息顺序的管道

上图描绘了一条非常基础的流。读取数据，然后验证、转换和写入。完全不使用 actor 便可以轻松地实现这个流程。每一步都可以实现成阻塞的操作，以便保证数据以读取顺序被写入。当然，这样不能公平地分享系统资源，如果正在等待写入操作的完成，则会因为它被阻塞导致不能进行其他操作。

但是，如果将每一步都实现为一个 actor，便会显现一些有意思的改进。因为这条流上并没有分支点，所以仍然可以像没有 actor 一样保证顺序。在将当前消息传递到验证阶段之前，读取阶段不能处理任何其他消息。在将消息传递到转换阶段之前，验证阶段也不能处理任何消息。但是由于这些 actor 的存在，读取、验证、转换和写入阶段都可以同时处理消息。读取阶段可以接收一个消息，将其传递给验证阶段后立即处理下一个消息，不需要一直等到先前的消息被传递到写入阶段。如果写入阶段需要很长时间才能处理完成，消息可以在队列中排队，直到上一条消息的写入操作完成，这也意味着写入阶段可以尽可能快地开始处理下一条消息，而不需要等消息被彻底处理完。

这个典型的例子说明，估算延迟时很容易被假象欺骗。如果要测量每条消息在整个流中的延迟时间，则每条消息被完全处理的时间等于每个阶段花费时间的总和，如下所示。

```
Latency = timeToRead + timeToValidate + timeToTransform + timeToWrite
```

当测量每个阶段的时间时，情况如下。

timeToRead	timeToValidate	timeToTransform	timeToWrite
2 ms	2 ms	6 ms	10 ms

通过方程可以确定延迟为 20 毫秒。然后，可以推断系统每 20 毫秒传递一次新的结果，每秒可以处理 50 条消息。实际上这样估算是错的。虽然单线程系统确实是如此操作，但这不适用于使用 actor 构建的系统。在 actor 系统中必须考虑到每个阶段可以并行发生的事实。系统的吞吐量受到流中最慢的阶段（在这种情况下，是写入阶段）的限制。这意味着吞吐量最终可能是每秒 100 个消息或每 10 毫秒一个消息。如果只测量延迟，就会忽略这个事实。

actor 方法的主要优点之一是，它在系统中引入了并发机制，不牺牲系统的确定性而保证流两端的消息是有序的。

这样的流是用于优化 Akka 应用程序的强大工具，允许我们充分利用现代的多核系统。但是，数据流无法解决我们遇到的所有扩展性问题。虽然这样的流可以提高吞吐量，但不能无限提高下去。而且这里有一些潜在的成本，虽然使用 actor 的方法比使用阻塞函数的方法更能降低总的消息处理时间，但实际上每一个阶段每次仍然只能处理一条消息，其他消息必须在队列中等待，并且该队列中的等待时间可能是无限的。因此，尽管流是重要的工具，但依然需要结合其他工具来实现高可扩展性。

路由器

虽然流可以提高系统的吞吐量，但流的能力受到系统操作的阶段数以及每个阶段需要耗费的时间限制的影响。另外，在某些情况下使用 future 可能会更有效率，因为 future 有可能产生更高的吞吐量。但是，如果想通过使用 actor 实现同等水平的吞吐量呢？可以做到吗？

为了将吞吐量提高到一条流以上的水平，很明显我们需要同时处理多条流，这就是路由器（routers）可以发挥作用的地方。如图 5-2 所示，路由器只是一个可以接收消息并将其路由到其他 actor 的 actor。路由器非常重要，因为它提供了扩展流的方法，从而可以超越一条流能够达到的吞吐量水准，但是这要以引入非确定性为代价。

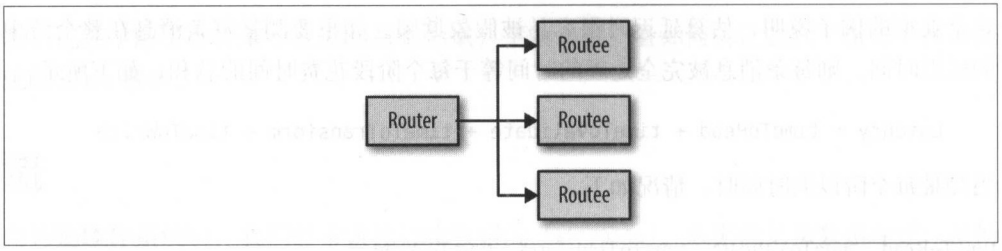


图5-2　路由器

Akka 提供了许多不同类型的路由器，但是它们的主要用途都差不多。路由器接收单个消息并将其发送给一个或多个 actor。至于发送给哪个 actor 或者发送给多少个 actor 主要取决于路由器的具体实现。如果现有路由器不足以满足应用程序的需要，也可以简单地创建自己的 actor 来封装所需的路由逻辑。

因为路由器的分支性质，系统将失去流提供的确定性。当通过路由器发送消息时，不能保证第一条消息就是第一条被成功发出去的消息。根据其路过的路由以及线程分配情况，

消息可能会以不确定的顺序离开系统。所以这里的诀窍是，在消息的顺序性不那么重要但是使用流的功能比较重要的场景下才使用路由器功能。

通常，在构建一个顺序很重要的系统时，会发现其实只是系统的某些局部对顺序的要求比较敏感。例如，在示例项目管理域中，只有某些特定请求需要按顺序处理。如果试图安排某人到某个特定项目，但是这些信息却没有按正确的顺序被处理，就可能会导致灾难性的后果。我们可能已经将某个不可用的人安排给了某个项目，或者可能会同意无法实现的要求。所以在应用程序的某些地方顺序处理消息是很关键的。另一方面，系统中也可能存在对顺序处理消息并没有特殊要求的地方。一个非常简单的例子是，我们要求每个单独的用户都按顺序处理一些消息，但是并不需要在多个用户之间按顺序处理这些消息。这种情况下，可以设计一个在单个用户中提供顺序保证的流，同时使用路由器来并行处理多个用户请求，如图 5-3 所示。

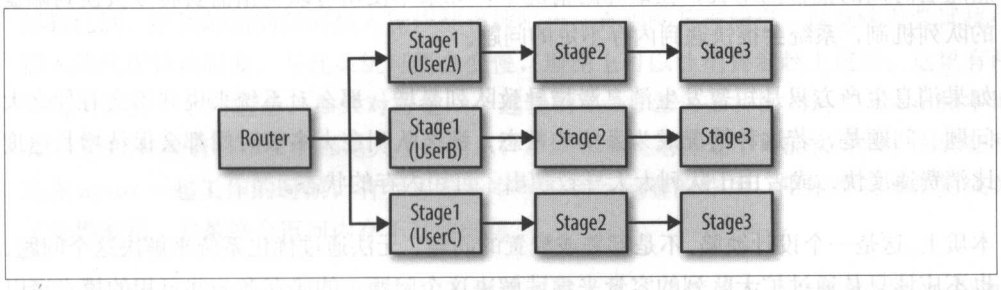


图5-3　路由器和流

上图中，我们创建了一个路由器，可以将消息传递给三个流之一。UserA 的所有消息都转到第一个流，UserB 的所有消息都转到第二个流，UserC 的所有消息转到第三个流。可以使用具有一致性哈希算法的路由器来创建此设置，也可以使用自定义路由器来执行此操作。使用 Akka 集群分片时，这样的设置也很常见，稍后将讨论。因为特定用户的所有消息都将转到同一个流，因此可以在该用户内维护消息的顺序。然而，多个用户之间的顺序并不重要，所以可以利用路由器来进行缩放。

有了这个新的结构，便可以同时处理更多的消息了。虽然每个流有三个阶段的限制，导致每个流只能处理三个并发消息，但是同时可以处理的流变得更多了。在图 5-3 中，可以同时处理的消息多达九个（三个流 × 三个阶段），并且不牺牲应用程序对顺序的要求。

使用路由器的另一个好处是，单个流中包含的多个阶段可以同时执行多个任务，前提是这些任务的顺序并不重要。比如我们可能需要确认某个员工在某段时期内是否可用，这需要单独检查这个员工每一天的行程安排然后把结果汇总。在这个例子中，先检查哪一天的行程并没有关系，同时处理所有的日常也是可以的。这就是一个使用类似路由器的

“分支”和执行检查的例子。反过来也可以改善端到端延迟，因为某些操作可以同时运行，因此减少了总时间。

路由器是很关键的部分。无论是使用 Akka 路由器还是简单地实现自己的 actor，将消息传递到其他 actor 时都需要使用路由器，不仅可以横向扩展，还可以纵向扩展。路由器可以更充分地使用系统资源，更重要的是减少可能发生的瓶颈，它是提高吞吐量和减少延迟的工具，因此对于构建高度可扩展的系统至关重要。

邮箱

在关于流和路由器的讨论中，我们一直忽略了一个问题：慢消费者问题。当系统产生的消息比系统可处理的消息要多时，就会出现慢消费者问题。自然的结果是，消费者的邮箱随着时间的推移而增长，在最坏的情况下，如果不使用可以根据需要将分页换到磁盘的队列机制，系统会很快遇到内存不足的问题。

如果消息生产方只是短暂发生消息激增导致队列暴增，那么对系统来说并不会有什么大问题。问题是，若这种情况成为系统的常态，那么队列在大多数时间都会保持增长速度比消费速度快，或者由于队列太大导致超出了可用内存的状态。

本质上，这是一个设计缺陷，不是部署或配置的问题。无法通过优化系统来解决这个问题，也不应该只是通过扩大队列的容量来尝试解决这个问题。的确存在一些可用的模式可以解决这个问题，但在讨论这些模式之前，首先来看看邮箱的细节以及它们可能提供的帮助。

邮箱通常可以分成两个类别：有界邮箱和无界邮箱。系统默认的是无界邮箱。

无界邮箱

无界邮箱是 Akka 默认提供的邮箱机制。如果能正确使用，无界邮箱对于大多数场景来说都是足够的。但是，如果使用的方法不当，很可能会导致内存不足。如果 actor 不能足够快地处理消息，消息则会被一直挤压在邮箱中，直到系统内存不足。

内存不足的问题不仅会导致 actor 终止，而且还会拖垮整个应用程序。显然这不是我们所希望的。程序设计者已经开始担忧这类无界邮箱了，因为它可能导致各种问题，所以大家认为使用它是危险的，但它却是默认的邮箱机制，这是为什么？为什么要默认选择可能会导致整个系统崩溃的邮箱机制呢？不应该选择一种可以防止此问题的邮箱吗？

实际上罪魁祸首并不是无界邮箱。例如一直向水桶里面注水，最后水桶会被填满然后水会溢出。但这是水桶的问题吗？或者是因为系统没有及时检查溢出吗？如果有特定的人

或机制可以在水桶达到极限容量的时候关闭水流，会不会更好？下面来讨论如何实现这样的系统，在这之前先介绍另一种邮箱类型。

有界邮箱

使用有界邮箱看起来是一个可以解决内存溢出问题的方案，在正确的使用场景下的确是。但是，与无界邮箱一样，我们需要了解其工作原理，以便在适当的情况下使用它们。使用有界邮箱可以设置容纳消息数量的上限，当消息数量超过限制时，邮箱将执行某些操作来缓和问题。具体操作取决于邮箱的类型，可分为两类：阻塞型有界邮箱和非阻塞型有界邮箱。

旧版本的 Akka 中存在阻塞型有界邮箱，但在最新版本中已被移除。它们被移除的主要原因之一是存在误导开发者的问题和潜在的系统风险。阻塞型有界邮箱的原则是，如果邮箱已满，则该邮箱的任何插入操作都将被阻止，直到邮箱被清空。这意味着尝试执行插入的线程将被阻塞，并且系统可能会变慢，理论上可以让消费者赶上进度。这里有两个问题。第一个问题是，如果对方是一个远程的 actor 怎么办？因为远程 actor 和本地 actor 之间并没有用于交流邮箱大小的消息，所以阻塞发送方是不可能实现的。因此在和远程 actor 一起工作的时候，有界邮箱基本起不到它该有的作用。阻塞型有界邮箱成为了无界邮箱，并最终会返回内存不足的问题。

第二个问题有点微妙。假设我们使用阻塞型邮箱，进一步假设有一个四线程的线程池，所有的 actor 都在该线程池中运行。现在有四个 actor 都尝试向邮箱已满的慢消费者发送消息，这时它们的线程将被阻塞并等待，同时，慢消费者被期待可以赶上进度。但是慢消费者在同一个线程池中运行：一个线程池只有四个线程，所有线程都被阻塞。这意味着消费者没有线程可以操作，已经陷入了僵局。

更通俗地说，阻塞型有界邮箱的真正问题是，它在 actor 之间创建了一个同步的依赖关系。这违反了 Actor 模型的核心思想：Actor 模型通过异步消息传递使 actor 彼此进行通信。阻塞型有界邮箱打破了 Actor 模型的规则。因此，阻塞型有界邮箱被移除了。

那么非阻塞型有界邮箱呢？它们如何工作？要使非阻塞型有界邮箱正常工作，需要知道邮箱溢出时该怎么办。我们不能阻止和减缓生产者，也不能掩盖溢出，因为这样又会使邮箱被认为是无界的。因此，当邮箱溢出时，我们要丢弃消息，没有其他的选择。这样虽然会导致消息丢失，但请记住 Akka 最多只发送一次（at most once）的机制，这种机制非常适合丢弃消息的场景。如果需要更强的交付保证，需要使用 `AtLeastOnceDelivery` 等工具重新发送消息，稍后将详细讨论。

无界邮箱的缺点是导致内存溢出。有界邮箱的缺点是导致消息丢失。如果出现一个需求，

要求既不能发生内存溢出问题，也不能出现消息丢失问题，该怎么处理呢？

当然，缓解慢消费者问题的最佳方案是加快消费者的速度。如果可以横向扩展消费者（通过增加更多的消费者实例），那么消费者的慢消费问题根本就不会发生。若无法实现，则必须重新评估系统的设计和消息交付保证机制。可以调整系统（例如允许消息不按顺序处理），允许消费者进行扩展。

如果不能完全避免消费者的慢消费问题，则必须寻求替代模式来解决这一问题。

拉取的工作模式

解决慢消费者问题的一个方案是“扭转”参与者的工作模式，让消费者主动拉取消息，而不是被动推送消息。

在这种场景下，需要一个主 actor 去管理消息和一系列的 worker actor。主 actor 不做任何实际工作，相反，它只委派工作给 worker actor。消息流如图 5-4 所示。

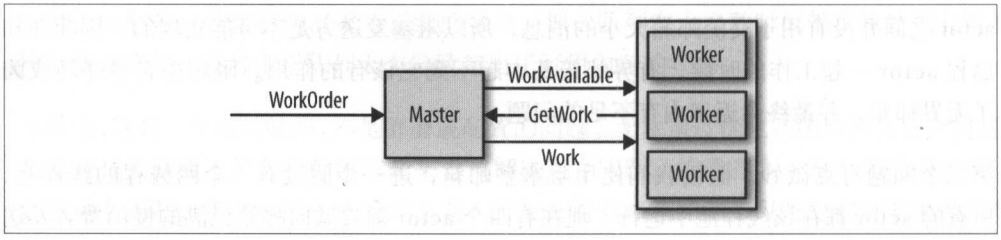


图5-4　拉取的工作模式

1．主 actor 接收到新的工作请求。

2．主 actor 通知一个或多个可用的 worker actor 准备开始工作（`WorkAvailable`）。

3．可以处理该消息的 worker actor 通知主 actor 已准备好，可以进行工作（`GetWork`）。

4．主 actor 将工作发送给可用的 worker actor 去处理。

这种模式的优点在于消除了 worker actor 邮箱溢出的可能性。worker actor 的邮箱在任何给定的时间内都只存有少量消息。

实现这一模式的关键在于，主 actor 要有减缓工作流的能力。这种能力具体表现为在需要时从文件或数据库中拉取工作，或丢弃传入的消息。如果主 actor 无法以这种方式控制工作流，那么刚刚的操作只是把问题转移到了别的地方而已：不是 worker actor 的邮箱发生了溢出的问题，而是主 actor 遇到了这个问题。

在构建分布式系统时，主 actor 经常被实现为一个集群的单例对象，以确保在任何给定的时间内只有一个主 actor。只要主 actor 能控制流量，系统就能很好地运行，但是如果主 actor 做不到，那么它会成为系统的瓶颈，我们需要考虑其他替代方案。

这种模式的一个很大的好处是，随着负载的增加或减少，可以很容易对系统进行扩缩操作。当新的 worker 上线了，它们会通知 master 自己的状态可用，然后 master 就可以向它们分配工作任务。当 master 发现所有的 worker 都超负荷工作时，它只需简单地扩展 worker 的数量即可。在分布式系统中尤其如此，在集群中增加一个新节点很容易，而且可以运行在不同的硬件上。

在任务调度的示例中，如果一个新项目被加入到系统中，则需要把这个项目分配出去。可以直接将消息推送到系统中，但调度是一个耗时的过程。因此，如果许多项目同时进行，可能会遇到邮箱溢出问题。为了解决这个问题，可以使用拉取的工作模式。创建项目时，将项目信息直接写入数据库。然后，项目调度程序将从数据库中提取这些信息记录，并向一系列 worker actor 发送消息。然后，每个 worker actor 将尝试安排该项目。代码如下。

```
object ProjectWorker {
  case class ScheduleProject(project: Project)
  case class ProjectScheduled(project: Project)
  def props(projectMaster: ActorRef): Props =
    Props(new ProjectWorker(projectMaster))
}

class ProjectWorker(projectMaster: ActorRef) extends Actor {
  projectMaster ! ProjectMaster.RegisterWorker(self)

  override def receive: Actor.Receive = {
    case ScheduleProject(project) =>
      scheduleProject(project)
      projectMaster ! ProjectScheduled(project)
  }

  private def scheduleProject(project: Project) = {
    // perform project scheduling tasks
  }
}
```

以上代码是 `ProjectWorker` actor 的简单实现。当一个 actor 上线时，它使用 `RegisterWorker` 消息向 `ProjectMaster` 申请注册。如果收到 `ScheduleProject` 消息，它将执行必要的调度任务，然后将 `ProjectScheduled` 消息返回给 `ProjectMaster`。该 `ProjectScheduled` 消息是给 `ProjectMaster` 的一个提示，提示 worker actor 完成了任务，并准备好接受更多的任务，代码如下。

```
object ProjectMaster {
  case class ProjectAdded(projectId: ProjectId)
  case class RegisterWorker(worker: ActorRef)
  private case class CheckForWork(worker: ActorRef)

  def props(projectRepository:
    ProjectRepository, pollingInterval: FiniteDuration): Props = {
    Props(new ProjectMaster(projectRepository, pollingInterval))
  }
}

class ProjectMaster(projectRepository:
  ProjectRepository, pollingInterval: FiniteDuration) extends Actor {
  import ProjectMaster._
  import context.dispatcher

  override def receive: Receive = {
    case RegisterWorker(worker) => scheduleNextProject(worker)
    case CheckForWork(worker) => scheduleNextProject(worker)
    case ProjectWorker.ProjectScheduled(project) =>
      scheduleNextProject(sender())
  }

  private def scheduleNextProject(worker: ActorRef) = {
    projectRepository.nextUnscheduledProject() match {
      case Some(project) =>
        worker ! ProjectWorker.ScheduleProject(project)
      case None =>
        context.system.scheduler.scheduleOnce(pollingInterval, self,
          CheckForWork(worker))
        self ! CheckForWork(worker)
    }
  }
}
```

在这个例子中，ProjectMaster 可以收到不同类型的消息，然而，行为都是一样的。如果收到一个 RegisterWorker 消息，它知道这是一个新的节点，在这种情况下会尝试安排一些新工作给这个新节点。如果收到一个 ProjectScheduled 消息，它知道一个工作节点刚完成一个任务。另外，ProjectMaster 将尝试找到更多的任务分配给这个节点。如果无法找到更多的任务分配，它将安排一个 CheckForWork 消息在过一段时间发送给自己。当 ProjectMaster 收到此消息时，便知道 worker 节点处于空闲状态，然后将再次尝试找到更多的任务分配给它们。

因为 ProjectMaster 只有在有工作节点空闲时才会拉取新的任务，所以这样就不会存在

邮箱溢出的风险。同时，ProjectMaster 的消息仅会被空闲的工作节点触发。这意味着在任何给定的时间点，ProjectMaster 邮箱中的消息数量都不会多于工作节点的数量，所以也不会有邮箱溢出的风险。

一个相关的拉取工作模式是使用外部排队系统，如使用 Apache Kafka 或 RabbitMQ 作为向工作节点分配任务的一种手段。这样可以在获得持久性队列或主题的同时保留并行性。可以直接在 Akka 中加入这样的队列，但通常使用现成的工具更为方便。

背压

拉取工作模式的替代方案是为消费者提供一种支持背压的方法。类似于有水流动的管道，这里则是指被数据填充的管道，系统对发送方施加压力以减慢流速，如图 5-5 所示。

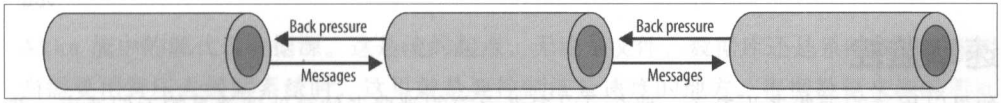


图 5-5　背压

当消费者数量达到系统的处理极限时，它会发出特殊形式的消息，或者可以由传输机制支持，提供透明的流量控制。

在任何一种情况下，消息都具有相同的效果：通知生产者减缓生产速度，以免压垮队列和消费者。

下面介绍几种背压方法。

ack

实现背压的一个简单的方法是让消费者给生产者发送一个确认消息，通常缩写为“ack”。例如，可以指定接收到了哪个消息，或者简单地说“OK，我已经处理了一个消息，可以再发送一个”。当然也可以应用于处理对象的数量多于一个的场景，例如消费者可以在处理一百条或一千条消息后发送确认，原理是一样的。

高水位标记

与基于 ack 消息的确认机制不同，另一个可行方案恰好相反。只有当生产者应该减慢或停止生产时才发送消息，一般是在超过所谓的高水位标记值（high-water mark）时。

正常的 ack 的意思是，已经处理完 X 条消息了，再发送一些吧。而高水位标记一般是指，

手上处理的事情很多，快处理不过来了，发送方需要放慢速度。

应该注意使用不同的通道来发送控制消息，因为通道上的积压可能会使控制消息的传递失效，导致更大的问题。

队列长度监控

跟踪生产者和消费者之间的队列长度是另一种可行的选择，事实上，这也可以用前面介绍的高水位标记法来实现。

需要谨慎使用这种方法，因为监视队列可能会给系统增加额外的负担，这是一个影响系统性能的方式，特别是在生产者和消费者不是单一组件的时候，系统中可能会有很多生产者和消费者，并且它们可能会被网络分离在不同的地方。

速率监控

如果以用过的消费者为参考，那么可以参考之前的消费者估算处理消息的近似速率，并使用该近似速率作为让生产者控制生产速度的参考。

这种技术的核心是，如果消费者可以在 Y 时间内处理 X 个消息，那么生产者在 Y 时间内不能发送超过 X 个消息。假设所有消息被处理的时间都很接近，这是一个不需要生产者和消费者之间直接沟通的简单的技术手段。

然而，像所有基于时间的操作一样，它会随着时间的推移而发生变化，这取决于整个系统的负载，所以条件改变时这种方法会变得不稳定。假设在消费者的节点上开始运行一些新进程，我们无法以以前的消费速度去处理消息，那么生产者发送消息的速度自然又会比消费者处理消息的速度更快了。

当然，可以建立自调整系统，其中消费者向生产者定期发送消息，不时地更新其速率信息。这样虽然复杂，但非常灵活，而且消费者和生产者之间的沟通相对较少。

所有这些技术的问题——无论是拉取的工作模式、背压、速率监控还是其他技术——都是需要额外的工作量的，它们都不是内置于系统中的。当建立纯粹的基于 actor 的系统时，需要结合这些技术来防止内存不足问题，但如果有内置的方法可以帮助我们，那将是更好的。

Akka 数据流

大家已经知道流如何提高消息吞吐量，以及路由器如何根据情况改善吞吐量和延迟，还知道邮箱如何溢出以及如何通过背压技术来解决这些问题。这些方法的演变结果就是我

们将要介绍的 Akka 流。Akka 流是 Akka 实现的响应式流，它采用了流、路由器、背压等概念，将其转换为单一一致的域特定语言（DSL），允许编写类型安全的流，如交叉点（junction point），也支持背压。它可以构建复杂的数据流，而不必担心 actor 的邮箱溢出。同时，它能提供与采用路由器作为分支点创建的 actor 流一样的所有优势。

Akka 数据流建立在几个基本概念之上，包括源（source）、汇（sink）、流（flow）和交叉点（junction）。当构建数据流时，它们中的每一个会起到一定有作用。我们将详细介绍这几点，以便大家能够充分了解它们是如何解决问题的。

为了可视化它们的作用，可以将 Akka 流设想成通过一套管道的水流。下面来看一下它们是如何工作的。

源

Akka 流中的源代表数据源。这是流的起点，无论是文件、数据库还是系统的其他输入。当需要用背压去缓解系统时，这里就是要控制流量速度的地方。根据数据来源的不同，可以采用不同的方法来减缓系统流速。可能是减慢从文件中读取数据的速度，也可能是减慢将数据缓冲到磁盘的速度，还可能是直接丢弃数据。一切都取决于源的性质和速度控制程度。

继续管道的类比，这里的源是水的起源。无论是来自河流、湖泊还是水库，在某些时候，我们需要从该水源中抽出水并将其通过管道泵送出去。想象一下，有一个巨大的湖泊，要从那里吸水。我们将一个泵放在湖里，开始连接管道，如图 5-6 所示。泵是水的来源，可以通过调节泵上的流速来加快或减慢水流过系统的速度。

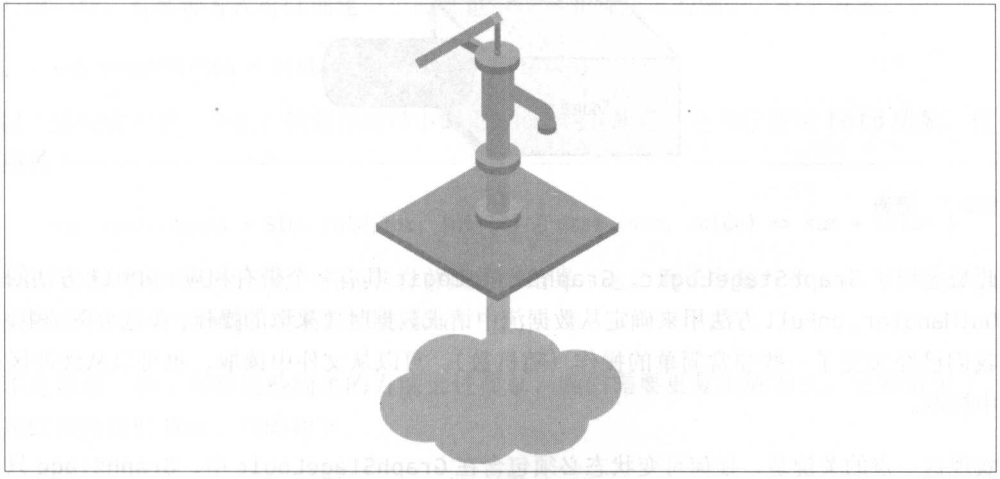


图5-6　管道源

有许多不同的方式可以构建源，可以通过使用简单的迭代器来构造，代码如下。

```
val positiveIntegers = Source.fromIterator(Iterator.from(1))
```

并不是总有以上这样简单的源，有时需要更复杂的构建方式。有各种不同的可用技术可以实现定制化的源，来看以下代码。

```
class RandomIntegers extends GraphStage[SourceShape[Int]] {
  private val out: Outlet[Int] = Outlet("NumbersSource")
  override val shape: SourceShape[Int] = SourceShape(out)

  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = {
    new GraphStageLogic(shape) {
      setHandler(out, new OutHandler {
        override def onPull(): Unit = {
          push(out, Random.nextInt())
        }
      })
    }
  }
}
```

GraphStage作为一个源还将配有一个出口。这个出口是源的输出，它连接到流中的其他元素。如果源是泵，则出口是连接到管道的连接器。这个出口会被传到一个shape上，在这个例子中是一个SourceShape。根据实际情况，也可以使用其他形状。SourceShape是一个简单的shape，只有一个出口，如图5-7所示，但如果需要，也可以创建具有多个出口的shape。

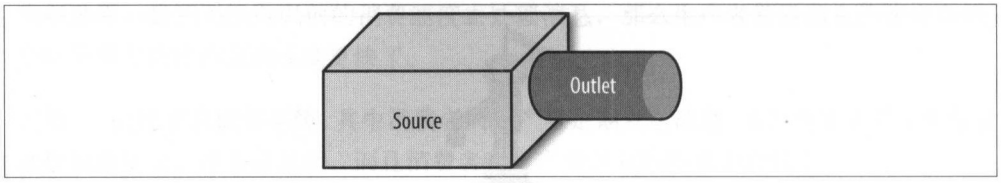


图5-7　管道

此处应用了GraphStageLogic。GraphStageLogic具有一个带有相应onPull方法的OutHandler。onPull方法用来确定从数据流中请求数据时要采取的操作。在这个例子中，我们已经实现了一些非常简单的操作（随机数），可以从文件中读取，也可以从缓冲区中拉取。

实现这一点的关键是，任何可变状态必须包含在GraphStageLogic中。GraphStage只是一个逻辑的模板，它是可重用的，可以在多个流中使用。有时我们可能不想在这些流

之间共享状态，这时 GraphStageLogic 将在每次创建流时被创建。这意味着每次将源连接到汇时，我们将获得一个新的 GraphStageLogic 实例。

汇

Akka 流中的汇表示流的终点。所有的流都会在某一个点结束。没有终点，流就没有真正的目的地。无论是写入数据库、生成文件还是在用户界面中显示某些内容，流的目标是产生输出。事实上，一个流只需要两部分：源和汇。

继续用管道进行类比，可以想象：从湖中抽水的泵将把水输入管道，房子里的用户打开水龙头时，水会流入水槽，如图 5-8 所示。这就是水的终点。当然，假设房子里的水槽直接连接到湖中的水泵，这中间没有任何东西，这么假设显然是不合理的，现实世界中的管道不会这样工作。

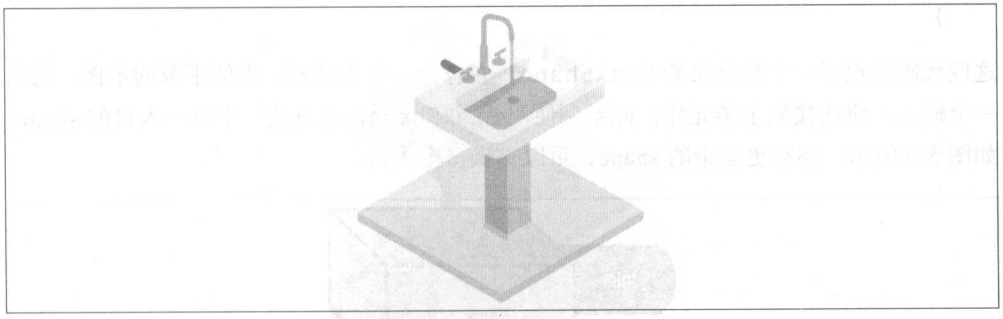


图5-8　管道汇

与源一样，有多种方式可以创建一个汇，也有很多非常简单的例子，如下所示。

```
val printString = Sink.foreach[Any](println)
```

以上语句将创建一个汇，只需在项目中调用 `println` 即可。也可以使用 `fold` 函数，代码如下。

```
val sumIntegers = Sink.fold[Int, Int](0) { case (sum, value) => sum + value }
```

`fold` 函数将处理每个元素，并将其合并成特定结果：这个例子中是一个 `sum`。`fold` 的结果将是 `future [Int]`，它将解析为 `sum`。

不过像源一样，有时这些简单的方法太过简单，我们需要更复杂的方式。这种情况下，可以回到图形 DSL，代码如下。

```
class Printer extends GraphStage[SinkShape[Int]] {
  private val in: Inlet[Int] = Inlet("NumberSink")
```

```
    override val shape: SinkShape[Int] = SinkShape(in)

    override def createLogic(inheritedAttributes: Attributes): GraphStageLogic = {
      new GraphStageLogic(shape) {
        override def preStart(): Unit = {
          pull(in)
        }

        setHandler(in, new InHandler {
          override def onPush(): Unit = {
            println(grab(in))
            pull(in)
          }
        })
      }
    }
  }
```

这段代码实现了一个自定义的 SinkShape。它包含一个 Inlet，类似于源的示例，表示一个输入，将连接到上游元件。此示例使用 SinkShape，它只是一个单一入口的 shape，如图 5-9 所示，还有更复杂的 shape，可以提供多个入口。

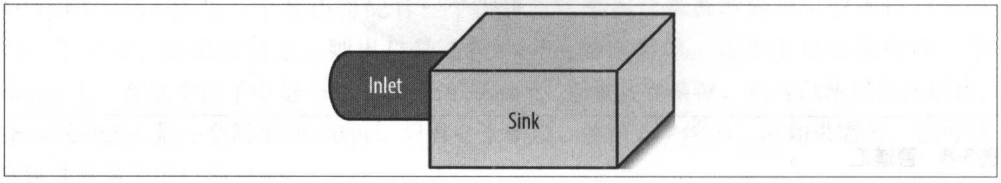


图5-9 汇

这里的逻辑非常简单：一个 InHandler 及相应的 onPush 方法。当数据被推送到流中时，会调用 onPush 方法。要访问该数据，可以调用 grab 来返回下一个元素。当准备好要求更多数据时，可以调用 pull。该操作会向上游元素发出信号，表明自己已准备好接收更多数据。

要启动数据流，需要表明自己的需求。可以通过重写 preStart 方法并调用 pull 来实现。

RunnableGraph

RunnableGraph 是将源和汇连接在一起的结果。这两部分可以组成一个最小的完整系统。有了 RunnableGraph 后，数据可以开始移动。必须同时具有源和汇，不然无法实现任何操作。缺少数据源点或是缺少数据终点的系统是不完整的。

每个出口必须正好连接到一个入口，如图 5-10 所示。如果任何入口或出口没有对接上，那么系统就是不完整的，也无法正常工作。

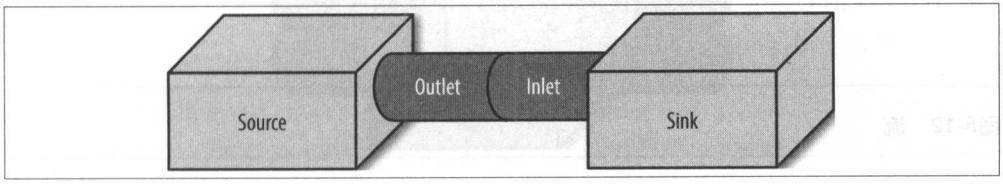


图5-10　RunnableGraph

从管道的角度来看，当将水源连接到水槽时，会生成一个 RunnableGraph，如图 5-11 所示。这样，水可以从一个地方自由流动到另一个地方，否则水泵抽出的水就不知道会流到哪里，水槽也不会有水流进来，除非将水源与水槽对接起来。

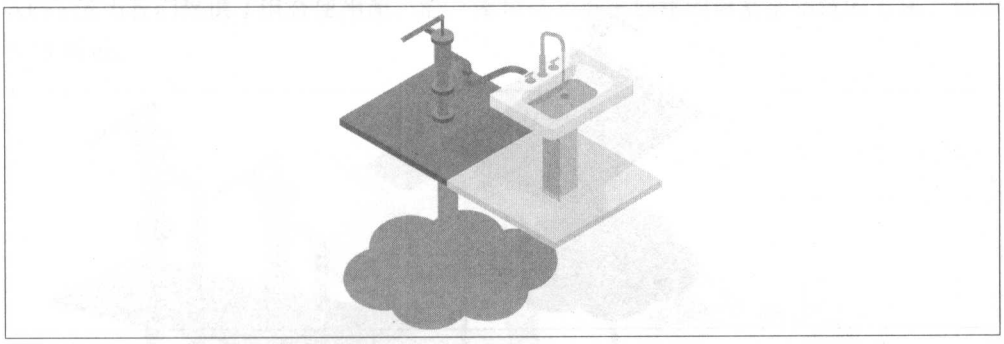


图5-11　管道系统

流

如前文所述，简单地连接源和汇就可以创建一个可运行的流，但这样的系统并没有什么实际用处。只是从一个地方拿走数据，把它放在另一个地方而已，没有任何优化或改变。有时候这也足够，但是大多数时候还是要对这些数据进行一些操作处理，例如以某种方式转换它，这也是流的目的。流是一个“连接器”，可以转换数据，如图 5-12 所示。流有许多不同形式，有些可能只是用来修改数据的，其他的可能会通过过滤数据或选择其中的一部分来减少数据。流连接了源和汇，它们采用了基本的 RunnableFlow 并增强了流，给它带来更多的功能。

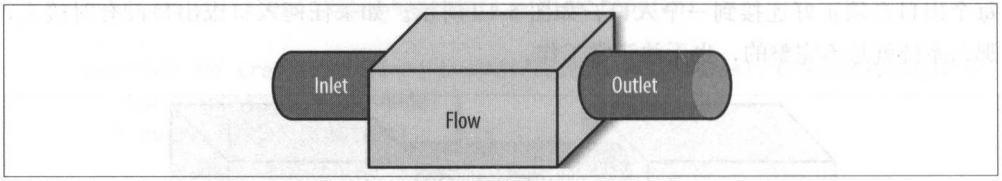


图5-12　流

回顾管道的例子，流就是管道。我们有一个水源，有一个水槽，但认为可以用固定长度的管道去简单地连接水源和水槽的想法是不切实际的。通常，我们需要将水输送一定的距离。如图 5-13 所示，输送过程可能需要更改几次路线。为了改变压力，管道可能变窄或变宽，通过使用各种形状和尺寸的管道来完成输水操作，每一段这样的管道都代表一个流。

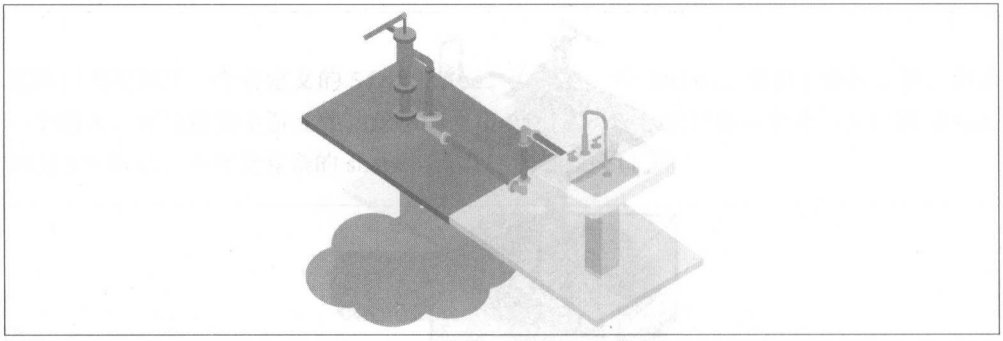


图5-13　管道流

交叉点

当数据在系统中流动时，可能需要有一个分岔口让部分数据通过另一条不同的路径。或者，可能需要从多个来源获取数据并以某种方式将它们汇集在一起。这时可以通过使用交叉点来实现上述操作。一个交叉点基本上就是一个分支点，它可以是扇入或者扇出的。通过交叉点可以将一个流分散到多个流中，也可以将多个流汇集成一个流，如图 5-14 所示。开始使用交叉点便意味着已经不再只是处理简单的流了：我们正在处理复杂的网络图。

交叉点被创建为具有多个入口或多个出口的图形元素。

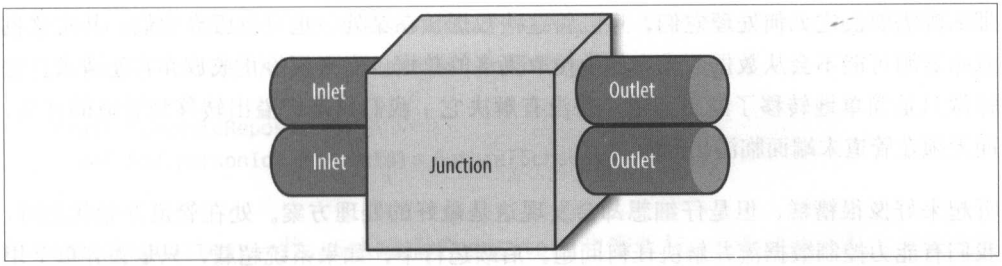


图5-14　交叉点

在管道的例子中，交叉点表示如T型接头或集合管一类的东西。它们从源头上获取水流然后分流到多个不同的房间内，每个房间都有自己的水槽。或者，它们可能会将热水和冷水汇集到一起为我们提供温水。

Akka流为我们提供了组合使用源、汇、流和交叉点来创建图形复杂系统的方法，如图5-15所示。

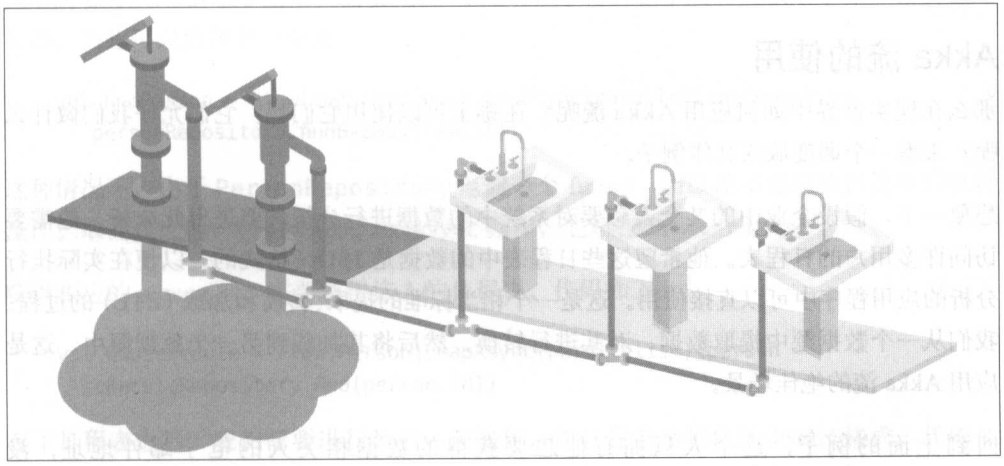


图5-15　复杂网络图

Akka流中的背压

Akka流建立在背压的概念之上。背压是通过push–pull机制实现的。当源和汇连接起来形成一个可运行流时，消息会由汇反向发送给源。这类消息基本上是为了告诉源，作为接收方的汇可以接收更多的数据。当源接收到该消息后，它便知道可以发送更多的数据给汇。当数据可用时，源便会将数据通过流发送给汇。

源只可以发送由汇请求过的数据。如果源获取了一些数据，但是并没有汇向源请求它们，

那么源需要决定如何处理它们，可以将这些数据放在某处，也可以丢弃它们。其实这也意味着源可能不会从数据库或文件中拉取更多的数据，它必须停止或放弃其他请求。这样做只是简单地转移了溢出问题，并没有解决它；我们只是把溢出转移到管道的开头，而无须在管道末端面临溢出问题。

听起来好像很糟糕，但是仔细想却会发现这是最好的处理方案。处在管道开始状态时，我们有能力控制数据流并解决任何问题。后续运行中，如果系统超载，只是丢弃单个用户的请求会更好，接受所有请求反而会导致系统内存不足，导致整个系统崩溃。在前一种情况下，只有单个用户以非常可控的方式受到影响。而在后一种情况下，所有用户都可能受到影响，并且影响程度非常不确定。

某些时候，由于硬件条件的限制，无论如何都无法接收处理完所有的数据。这时候需要给系统的输入端施加反向压力来减缓数据输入的速度，这就是 Akka 流可以做到的。它迫使压力回到系统的入口，提供必要的机制以便在系统遇到负载超载时可以通知源放慢数据输入，这也是真正处理溢出问题的唯一方法。

Akka 流的使用

那么在现实世界中如何应用 Akka 流呢？在哪里可以使用它们呢？它们允许我们做什么呢？来看一个调度域的具体例子。

想象一下，假设企业中的某个人想要对系统中的数据进行分析。要进行此分析，他需要访问许多用户的日程表。他希望这些日程表中的数据是 JSON 格式的，以便在实际执行分析的应用程序中可以直接使用。这是一个相当标准的提取、转换和加载（ETL）的过程。我们从一个数据源中提取数据，对其进行转换，然后将其加载到另一个数据源中，这是应用 Akka 流的绝佳场景。

回到上面的例子，这个人只拥有他想要获得的数据相关人的电子邮件地址，没有对方的实际 ID，实际 ID 需要额外的查找才能得到。这种情况下可以通过使用 `PersonRepository` 来实现，代码如下。

```
trait PersonRepository {
  def findByEmail(emailAddress: EmailAddress): Future[Person]
}
```

这段代码将在其他服务的有界上下文内运行，所以 `PersonApi` 需要以某种方式与其他有界上下文进行通信。这时我们可能想要使用 actor 系统集群的场景，或者可能会选择使用 Akka Http 通过 REST API 来对外暴露这些数据。在任何情况下，`PersonRepository` trait 都是一个隐藏实际实现细节的绝缘层。

有了 Person 后，我们可以获取 PersonId，然后使用它来查找对应的 Schedule 的详细信息。为此，我们将使用一个 ScheduleRepository，如下所示。

```
trait ScheduleRepository {
  def find(personId: PersonId): Future[Schedule]
}
```

与 PersonRepository 一样，ScheduleRepository 也需要与一些单独的有界上下文进行通信。该上下文可能与 PersonRepository 涉及的上下文不一样。毕竟，调度无须知道电子邮件的地址。然而，对于正在建立的数据分析报告而言，我们需要从两个环境中分别获取信息。生成此报告将从必要的数据源拉取数据，然后将其转换为所需的 JSON 格式。

输入的数据将是一系列字符串形式的电子邮件地址。然而，trait 需要使用一个代表电子邮件地址的 case 类，因此需要执行转换。可以创建一个实现此操作的流，代码如下。

```
val toEmailAddress = Flow[String].map(str => EmailAddress(str))
```

获取到 EmailAddress 之后，需要查找 PersonRepository 确认与该电子邮件相关联的人员。为此可以使用另一个流。

```
val findPerson = Flow[EmailAddress].mapAsync(parallelism)(email =>
  personRepository.findByEmail(email))
```

这种情况下，由于 PersonRepository 返回一个 future，所以流不能只执行简单的映射操作。取而代之地，我们将使用 mapAsync 操作，它将在 future 上运行。

获得相应的 person 后，需要获得该人的日程表，代码如下。

```
val findSchedule = Flow[Person].mapAsync(parallelism)(person =>
  scheduleRepository.find(person.id))
```

有了日程表之后，我们需要进行最后一项操作，将日程表序列化为 JSON 格式。代码如下所示。

```
val toJson = Flow[Schedule].map(schedule => serializer.toJson(schedule))
```

当所有的代码片段都到位后，可以将它们整合到流中。

```
Source(emailStrings)
  .via(toEmailAddress)
  .via(findPerson)
  .via(findSchedule)
  .via(toJson)
  .runForeach(json => println(json))
```

在这个简单的例子中，我们只是打印了 JSON，当然也可以将该 JSON 发送到另一个 API 或将其写入文件中，还可以将其保存到数据库中。

系统将拉取电子邮件地址的字符串值，并通过完整的流运算将它们发送到目的地。这一切都以支持背压的方式完成，这样就不会让系统超负载运行。如果 `findSchedule` 流恰好是一个瓶颈，使用上述方式则可以避免它发生溢出问题，因为背压机制会阻止它。同时，我们可以利用系统的异步性质，当 `findSchedule` 方法等待返回时继续处理上游或下游的数据，继续执行更快的 `findPerson` 查找，以便在进行下一个 `findSchedule` 前准备好数据。只要有数据，我们也可以继续执行 `toJson` 操作。

某些时候，如果 `findSchedule` 进程太慢，有些操作会处于等待状态。比如 `toJson` 进程将处理完所有可处理的数据而等待，或者背压机制会迫使我们暂停处理 `emailStrings` 的操作而等待 `findSchedule` 进程赶上进度。这是慢操作的本质。但是直到发生这种情况，我们都可以充分利用可用的系统资源。

结论

本章介绍了如何将结构良好的 actor 集成到系统中来更好地支持数据流，同时维持背压和可靠性。在第 6 章中，我们将深入了解如何在正确维护数据一致性的同时不牺牲系统的可扩展性。

这是一个微妙的平衡，在很大程度上依赖于模型中的 actor 是否正确构造以及这些 actor 是否合理整合。

第 6 章

一致性和可扩展性

一致性是系统内比较复杂的属性，并且它会随着系统的变化而发生变化。这个概念经常被误解，所以首先来看一看一致性是什么意思，然后再研究如何控制它。

如果系统中不同的相关数据的状态是一样的，则可以说系统是具有一致性的。

举一个简单的例子，假设系统记录了我们在银行账户的存款总额以及提款总额，那么如果系统记录的银行余额与这两个数字之间的差额是一致的，则认为该系统是具有一致性的。

在经典的单处理器、单存储空间的冯诺依曼系统中，这种一致性并不难实现。当然，前提是假设系统没有任何 bug。

然而，一旦系统具有并行性，保持一致性就变得困难了。如果我们在系统中引入了多核，那么系统就有了并行性。如果让系统变成分布式的，情况则会变得更加复杂。

正如我们在第 1 章中所解释的，现实世界并不具有全局一致的状态，所以如果软件系统也不具有一致性，这是可以接受的。

在本章中，我们将讨论分布式系统上下文中的一个事务意味着什么，一致性需要权衡什么，为什么全局一致性对可扩展系统并不友好，还将讨论消息交付保证以及避免影响系统可扩展性的方法。

事务和一致性

一致性意义上的事务是指单一的原子操作。这意味着更改中涉及的值或全部更新，或者全都不更新——不存在只有部分值被更新的情况。当然，在分布式系统中，这可能是一个挑战。

其典型的例子是一个数据库事务，在这个事务中，几个表在一个原子操作中被更新。除此之外还有许多其他例子。

讨论一致性时，我们将看到它如何应用于 Akka。

强一致性与最终一致性

数据在任一时刻都是被原子性的事务操作所更新的，这种情况被称为强一致性，即整个系统的所有节点可能会从一个完全一致的状态转移到另外一个完全一致的状态，虽然这两个状态不同，但是整个系统所有节点都一直对同一个状态保持一致，并不会在某个时间点上出现节点之间状态不一致的情况。

另一方面，最终一致性意味着在一段时间内发生的更新不管跨度多短，系统的不同节点在这段时期内都会有不同的状态，但在一段时间后，节点间的状态最终会达到一致。

奇怪的是，最终一致性通常比强一致性更容易实现，特别是在分布式环境中。

并发性与并行性

并发性是指两个进程从某一个时刻开始同时存在，但它们不一定要在同一时刻运行，也可以在重叠的时间段内运行。也就是说，可以在一开始先运行某个进程，然后停止它，再开始运行另一个进程，运行一段时间，停止该进程再次启动先前的进程，在同一个时间段内一直这样反复下去，直到结束。并发性是指多个进程并不一定要在同一时刻运行，它们只是运行在重叠的时间段内。

然而，并行性引入了两个进程同时执行的意思。换句话说，即两个（或更多个）进程在同一时刻运行。支持并行性意味着也支持并发性，但反过来就不一定了：系统可以支持并发性但不支持并行性。

根据定义，分布式系统是支持并行性的，而单核心单处理器系统是支持并发性的，任务可以在同一时间段内交替执行直到所有任务都完成。

为什么全局一致的分布式状态影响可扩展性

全局一致的分布式状态除了违反现实世界中的因果法则之外，同时还很难实现。

抛开这种状态是否可取，我们先假设系统就是需要全局一致的分布状态。若更新状态，我们希望所有相关的节点在更新完成后（例如数据的副本或从其衍生出的一些数据，如银行卡余额）对于状态的更新是完全同步的。

这明显意味着，状态更新时，我们需要“停止整个世界”，在此期间检查系统的每个相关模块并将其更新到正确的状态，然后所有节点才可以继续工作。

一个节点在此过程中可能不可用，节点之间如何保证同步现在的时刻点（各个节点自身的系统时间可能会不一致，每个节点根据自己的系统时间来确定当前时间就会产生不一致的状态）？这也是一个问题，可能需要一本专门的书来介绍。

在分布式环境中执行此类操作会消耗整个系统的重要资源，随着分布式组件的增加，这种情况会变得更糟。同时，在更新状态期间，每个节点都必须避免在旧状态下进行相关的工作，直到它们都更新为最新的状态值，这严重限制了并行执行处理的能力，从而降低了可扩展性。

位置透明性

位置透明性也是系统的一个特征，通过它可以执行计算而不用关心计算发生的位置（例如节点）。

在 Akka 中，actor 之间的信息流对开发者来说也应该是位置透明的，也就是说，我们发送消息时不应该关心收件人是否也在同一个节点上这一问题。系统本身会根据需要将消息传递到正确的位置，无须开发者直接参与。

交付保证

交付保证是建立分布式系统时经常被忽视的一部分。在使用某些特定通信机制时，我们常常考虑交付保证，但在使用其他机制时又会忽略它们。当通过事件总线发送消息时，我们会考虑该事件总线提供的交付保证机制，但是当通过 HTTP 请求发送时，往往不会考虑关于交付保证的东西。假设消息将被传递，如果交付成功，将得到一个成功的回应，没有回应意味着消息没有被成功传递。然而，情况并非如此。

系统提供的交付保证很重要，因为它们可能会对该系统的一致性产生重大影响。有三种基本类型的交付保证，但实际上只有两种是可实现的，第三种仅可以通过使用某些技术来接近，我们将在后面探讨，先来看可实现的两种类型。

最多投递一次

最多投递一次的交付机制是最容易实现的。它不需要通信中的任一方用内存或者磁盘去存储消息本身。这种机制也意味着当我们发送消息时，虽然不知道是否会成功投递，但这是合理的。因为通过这种交付保证，我们必须接受消息可能丢失的情况，这种机制就

是这样设计的，允许丢消息。

最多投递一次的交付机制表示只需要简单地发送一次消息，然后就可以继续进行其他处理，不需要等待接收方返回的确认消息。这是 Akka 默认的交付机制。当使用 Akka 将消息发送给 actor 时，不能保证该消息被成功接收。如果 actor 是本地的，可以假设只要系统不失败，消息就将被成功送达对方的邮箱，但是如果系统在 actor 从邮箱中获取该消息之前发生了故障，那么该消息将会丢失。记住，消息处理是异步的，因此消息已经被发送并不意味着它已被送达并处理。

如果 actor 是远程或集群的 actor，操作会变得更加复杂。在这种情况下，需要处理远程 actor 系统崩溃的问题，还必须考虑由于网络问题而导致消息丢失的可能性。无论 actor 是本地还是远程，都不能保证消息将被成功送达。如果消息是否成功投递对于系统来说至关重要，那我们必须努力寻求替代的交付机制。

最少一次

最少投递一次的交付机制在任何系统中都比较难以实现。它需要存储消息本身以及来自接收方的确认消息。该存储可能在内存中，也可能在磁盘上，具体取决于投递消息的重要性。如果想让交付成功，需要确保其存储在可靠的位置；否则如果发送方出现问题，消息就可能会丢失。

无论消息存储在内存上还是磁盘上，最基础的处理步骤都是先存储消息，然后发送消息并等待确认。如果没有收到确认，则要重新发送该消息，继续等待，直到收到确认。

最少投递一次的交付机制可能会导致一条消息在接收方被重复接收。比如，如果确认信息丢失了，那么再次发送消息时就会产生重复发送该消息的问题。但是由于我们会一直发送消息直到收到确认，所以可以保证接收方至少会收到一次消息，因此这种交付机制是可靠的。

在 Akka 中，可以通过几种方式来实现最少投递一次的交付机制。第一种方法是手动实现。这种情况下，只需将消息存储在某个队列的某个地方，发送并等待响应。当收到响应时，在队列中删除该消息即可。我们还需要恢复机制，以便在未收到响应时可以重新发送消息。

可以使用 Ask 模式轻松地在内存中实现上述方法，代码如下。

```
class MySender(receiver: ActorRef) extends Actor {
  import context.dispatcher
  implicit val askTimeout = Timeout(5.seconds)
```

```
    sendMessage(Message("Hello"))

    private def sendMessage(message: Message):Future[Ack] = {
      (receiver ? message).mapTo[Ack].recoverWith {
      case ex: AskTimeoutException => sendMessage(message)
      }
    }

    override def receive: Receive = Actor.emptyBehavior }
  }
```

这是一个非常简单的例子：发送消息，在发生 Ask TimeoutException 的情况下尝试重新发送消息。当然，只有接收方没有崩溃，这种类型的交付才是可靠的。如果接收方崩溃，就无法提供交付保证了。

可以调整刚刚描述的解决方案，引入一些数据库或可靠的磁盘存储。事实上，Akka Persistence 内置了所有逻辑，并提供了 AtLeastOnceDelivery 特性，如下面的代码所示。

```
case class SendMessage(message: String)
case class MessageSent(message: String)

case class AcknowledgeDelivery(deliveryId: Long, message: String)
case class DeliveryAcknowledged(deliveryId: Long)

object MySender {
  def props(receiver: ActorRef) = Props(new MySender(receiver))
}

class MySender(receiver: ActorRef) extends PersistentActor
  with AtLeastOnceDelivery {
  override def persistenceId: String = "PersistenceId"

  override def receiveRecover: Receive = {
    case MessageSent(message) =>
      deliver(receiver.path)(deliveryId =>
        AcknowledgeDelivery(deliveryId, message))
    case DeliveryAcknowledged(deliveryId) =>
      confirmDelivery(deliveryId)
  }

  override def receiveCommand: Receive = {
    case SendMessage(message) => persist(MessageSent(message)) { request =>
      deliver(receiver.path)(deliveryId =>
        AcknowledgeDelivery(deliveryId, message))
    }
```

```
      case ack: DeliveryAcknowledged => persist(ack) { ack =>
        confirmDelivery(ack.deliveryId)
      }
    }
  }
```

这是使用 AtLeastOnceDelivery 机制的一个简单的 actor 实现。请注意，该实现使用的 AtLeastOnceDelivery 扩展了 PersistentActor。该 actor 将接收 SendMessage 形式的命令。当收到这个命令时，它会将消息发送给另一个 actor，但是这里需要保证消息被成功交付。通过在发送消息之前持久化消息可以保证成功交付，要使用 deliver 方法而不是标准的 tell 方法。使用 deliver 方法会生成一个交付 ID，该 ID 需要包含在传出的消息中（在本例中为 AcknowledgeDelivery）。

当接收方的 actor 收到包含交付 ID 的消息时，它会使用包含相同交付 ID（这种情况下为 MessageAcknowledged）的另一个消息来响应发送方。

```
sender() ! MessageAcknowledged(deliveryId)
```

发送方接收并持久化确认消息，同时确认交付，如图 6-1 所示。

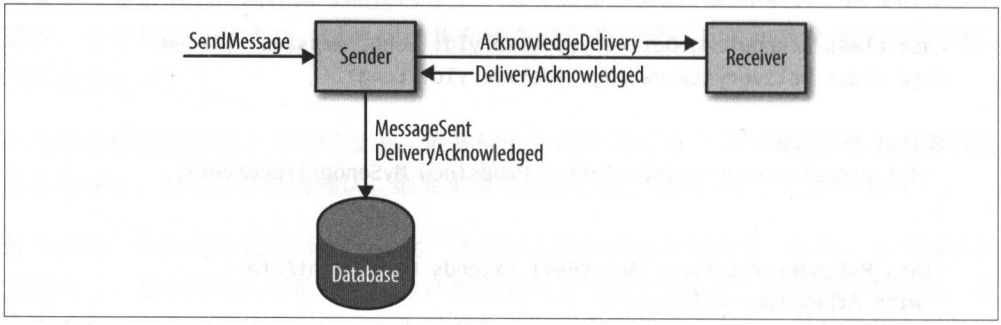


图6-1　至少投递一次的消息流

在图 6-1 中，可以看到发送方如何存储发出的消息，然后根据 DeliveryAcknowledged 对其进行检查确认，如果确认发出的消息被成功传递，就不需要再存储了。

在底层，actor 有一个可配置的定时器。如果在配置的时间内没有收到确认，它将自动尝试重新投递。

如果发送方出错怎么办？发生错误时，发送方会重新启动，重新加载持久化的消息以及相应的确认消息。不过与标准的持久化 actor 不同，它们会有一个缓存机制。在重启后，消息并不会被立即重新发出，而是先加载到缓存内。在所有消息都被重新加载后，发送方在缓存内逐一比较待发消息和已经确认的消息，它会跳过那些已经确认被收到的消息，

然后再重新发送那些没被确认的消息。

这意味着如果接收方发生错误或丢失确认消息，超时机制将确保重新传递该消息。如果发送方在收到确认消息之前发生故障，恢复时则会发现没有收到相应的确认消息，因此重新发送该消息。这样可以保证无论发生任何故障，消息都将被至少成功投递一次。

恰好一次交付是不可能的（但可以近似做到）

在很多情况下，我们需要的是恰好一次的交付机制，也就是希望每条消息恰好只投递一次就可以成功，不多也不少。不幸的是，这种类型的交付在任何系统中都是不可能实现的。只能结合使用一些可行的交付机制来平衡这种需求。

来看一个非常简单的例子。域中有多个有界的上下文，其中一个有界上下文是负责更新系统中人员信息的子系统，还有一个有界的上下文是负责管理这些人员分配的调度子系统。如果需要删除或停用某人（他已经换到另一家公司了），则需要同时更新这两个子系统。当然，我们希望两个系统就该人是否在系统中以及该人的分配情况达成一致。简单的解决方案似乎是，更新一个子系统时，该系统将相关消息发送给另一个子系统并告诉它执行相应的操作进行同步。

所以，更新人员信息的子系统在删除某人时需要向人员调度子系统发送消息以移除该人在人员调度子系统中的相关信息。重要的是，如果此消息已经发送，则需要等待对方确认。但是如果确认信息一直没有返回呢？当人员信息子系统试图发送消息时发生了网络分区问题又该怎么办？如何恢复？可以假设消息没有成功投递并再次发送。但是如果消息实际上已经被成功投递，只是返回的确认消息丢失了而已，则重新发送消息将导致重复交付。另一方面，如果假定该消息已经接被收，不需要重新发送，那么我们将承受消息没有被成功投递到对方的风险。这么看来，无论如何都没有办法保证信息恰好只被投递一次。

但是恰好一次的交付机制通常是我们想要的，因此需要找到一种可以接近它的方法。

如何近似做到恰好一次交付

许多开发人员和设计者认为他们想要的是恰好投递一次的交付机制，也就是说，消息从发送者到接收者只需要恰好一次可靠的传递，没有多余的重复。

实际上，恰好投递一次的交付机制仍然是不可能做到的，只是有可能接近它，但也只能通过一个潜在的可能机制来实现。该解决方案本质上是通过使用最少投递一次的交付机制来实现的，只是从上层看起来像是恰好投递一次的交付机制，至少从开发者的角度来看是这样的。

开发者会认为是以恰好投递一次的交付机制来发送消息的，并且依赖于底层机制来发送消息，接收确认，如果需要就重新发送，直到消息被确认接收，这其中重复的消息会被去重后再传送给接收方。这个过程可能会影响性能，特别是在进行确认之前必须进行多次重传。该方法还需要对消息进行持久化处理，至少发送方需要，不过通常发送端和接收端都需要对消息进行持久化处理。

通常人们希望对底层机制有更多的可见性，所以在 Akka 系统中看到一个对“恰好投递一次的交付机制”的抽象是相当罕见的。

集群单例

有时候，系统中有一些区域必须是唯一的，创建多个副本是不可接受的，因为系统的这些区域可能具有必须被严格控制的状态。一个典型例子就是唯一 ID 的生成器。唯一 ID 的生成器需要跟踪以前使用过的 ID，或者可能使用单调递增的数值。在任何一种情况下，都需要确保在给定时间内只发生一次生成 ID 调用，尝试并行生成 ID 可能会导致 ID 重复。

本例中的 ID 生成器其实是系统中的瓶颈，在条件允许的情况下应该尽可能避免使用。尽管如此，有时 ID 生成器却是必要的。这种情况下，Akka 提供了确保整个集群系统中只有一个 actor 实例可用的功能，这是通过 Akka 集群单例实现的。

集群单例一般都运行在集群中存在时间最久的可用节点上。通过 actor 系统的 gossip 机制可以确定哪个节点是最久的节点。当集群确定最久的节点后，actor 的单例可以在该节点上实例化。所有这些都是通过使用集群单例管理者来处理的。该管理者将负责确保单例运行的细节不出现问题。

围绕集群单例封装的 actor 叫作集群单例代理，集群单例代理主要的作用是和单例通信。所有消息都通过代理传递，代理的工作是确定向哪个节点发送消息。单例的存在对所有其他节点来说都是透明的，它们只知道一个代理实例的存在，对于底层的细节并不知道。

如果托管着集群单例的节点出现问题该怎么办？如果发生这种情况，集群单例就会在一段时间内是不可用的。那么集群将重新决定哪个节点是运行时间最久的节点，并重新在该节点上初始化一个集群单例。在集群单例不可用期间，发送到集群单例中的任何消息都将被缓存。一旦集群单例再次可用，消息将被重新传递。如果这些消息有超时时间限制，那么它们很可能在集群单例再次可用前就已经超时了。

单例模式的最大缺点是无法保证可用性。没有简单的方法可以确保每时每刻总是有可用的节点存在。系统中总会存在单例从不可用到重新可用的过度时期。即使这个时期在许多情况下可能很短，但它仍然是系统的潜在问题。

单例模式的另一个缺点是，它会成为系统瓶颈，因为系统中只有一个 actor 实例，一次

只能处理一条消息，这意味着系统可能会因为处理那些积压的消息而需要等待很长时间。

另一个潜在的问题是，当单例从一个节点迁移到另一个节点时，储存在该单例中的任何状态都将丢失。重新建立单例时也需要重新建立这些状态。使用 Akka Persistence 可以维持状态，这样当重新创建单例时，先前的状态将自动恢复。

由于集群单例模式存在上述缺点，因此建议除非绝对必要，不然尽量避免使用。当必须使用的时候，应该尽可能地让单例的影响范围缩小。例如，如果使用单例来生成唯一的用户 ID，则最好是只做这一件事情。尽管也可以让单例在数据库中创建用户并执行初始化用户状态的操作，但是单例需要执行的每一步额外操作都会造成更大的系统瓶颈。如果能把它限制在单个操作中（最理想的是限制在内存中），就能确保其尽可能保持可用状态。

以下是使用 Akka Persistence 实现的一个非常简单的 ID 生成器的示例。

```
object IdProvider {
  case object GenerateId
  case class IdGenerated(id: Int)

  def props() = Props(new IdProvider)
}

class IdProvider extends PersistentActor {
  import IdProvider._

  override val persistenceId: String = "IdProvider"

  private var currentId = 0

  override def receiveRecover: Receive = {
    case IdGenerated(id) =>
      currentId = id
  }

  override def receiveCommand: Receive = {
    case GenerateId =>
      persist(IdGenerated(currentId + 1)) { evt =>
        currentId = evt.id
        sender() ! evt
      }
  }
}
```

IdProvider 类非常简单。它接收一个消息 GenerateId，并返回一个结果 IdGenerated。

actor 本身在收到 GenerateId 命令消息后，将创建一个 IdGenerated 事件，持久化该事件后，更新当前 ID，然后将该事件发回发送方。

当重新创建 IdProvider 时，如果发生故障，它将使用 receiveRecover 行为回放所有先前的消息，恢复其进程状态，因此不会有任何消息丢失。然后，它将继续在恢复的节点上进行 ID 生成的操作，就好像没有中断一样。

这个 IdProvider 与 Akka 集群无关。无论是否在集群环境中，此代码都可以用于所有 actor 系统。这个例子很好地说明了位置透明性的好处。可以假设这是一个本地的 actor，然后建立系统，即使后续将它作为一个 actor 集群，代码也基本不需要改变。

在确定要将此 actor 集群化后，实际上用很少的代码就可以将其转换为单例。代码如下。

```
val singletonManager = system.actorOf(ClusterSingletonManager.props(
    singletonProps = IdProvider.props(),
    terminationMessage = PoisonPill,
    settings = ClusterSingletonManagerSettings(system)),
  name = "IdProvider"
)
```

以上是创建 ClusterSingletonManager 的示例。必须在可能要托管该单例的节点上创建此管理者。然后，它将确保在这个节点上创建单例。

不能通过管理者发送消息，若要向该单例所在的 actor 发送消息，需要创建一个 ClusterSingletonProxy。要创建代理，请使用以下代码。

```
val idProvider = system.actorOf(ClusterSingletonProxy.props(
    singletonManagerPath = "/user/IdProvider",
    settings+ = ClusterSingletonProxySettings(system)),
  name = "IdProvider")
```

代理会接收一个通向 actor 的路径参数。此处得到的是 ActorRef，ActorRef 将作为单例的代理。我们可以像使用系统中的其他 ActorRef 一样使用此 ActorRef。另外，这也表明了位置透明性的好处。任何以前向本地 ActorRef 发送消息的 actor 现在也都可以向此代理发送消息了，无须在意该代理现在是否是集群单例的 actor 也可以继续正常运作，就好像该代理 actor 是本地的，大部分逻辑代码都可以保持不变。代码中唯一需要改变的地方就是构造 actor 的地方，现在的逻辑是构建集群单例代理，而不是构建一个本地的 actor，仅此而已。

可扩展性

具备可扩展性的系统在处理更高的负载时不会出现故障或性能急剧下降的问题。它与性

能的关系微乎其微，优化性能不能保证可扩展性（反之亦然）。

提高性能意味着系统可以以更快的速度响应一定规模的负载，而可扩展性更关心系统对更高负载的反应，无论是同时发送更多的请求、更大的数据集，还是要求更高的请求速率。

图 6-2 给出了系统在负载持续增加的情况下响应时间的变化。该图显示，可扩展性差的系统将有一个拐点（从那个点开始会发生完全或部分错误），有一些请求不会被处理，或者其性能会迅速下降。也就是说，每个请求的响应时间都随着负载的增加而迅速增加。当响应时间太长时，客户端会超时。

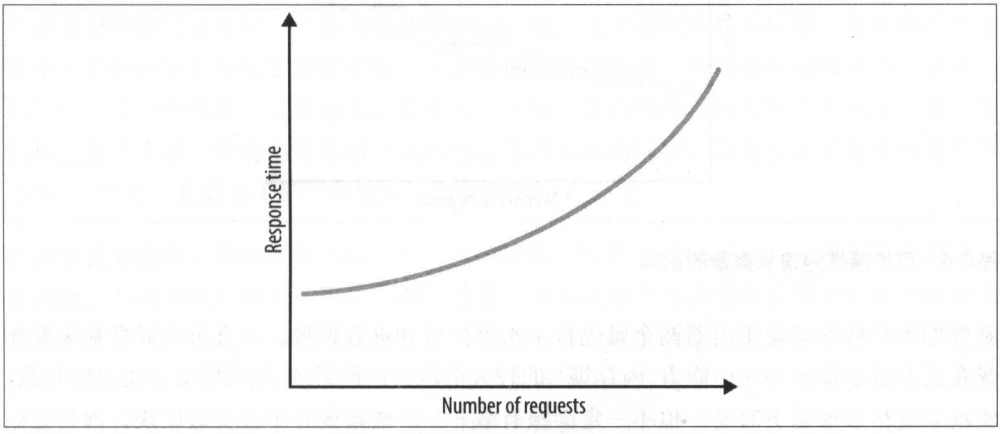


图6-2　响应时间与请求数

提高系统的性能后，系统响应时间线将向下移动（响应更快），而其他指标不移动，如图 6-3 所示。

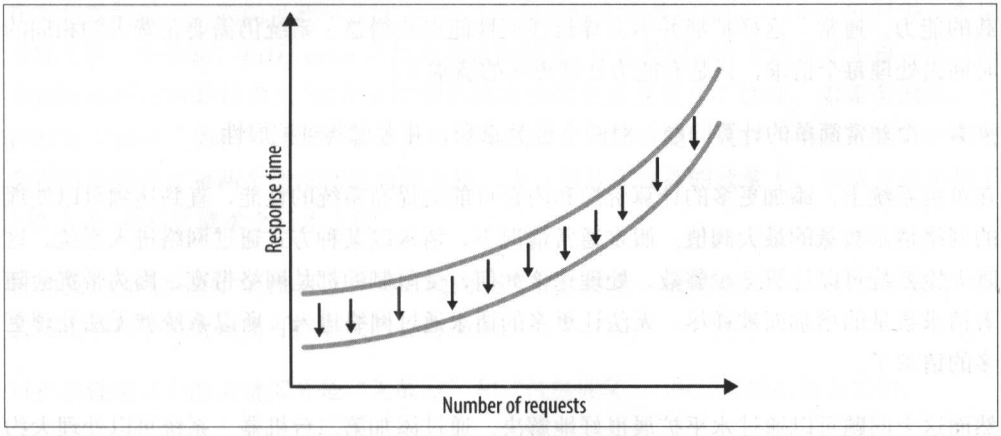


图6-3　性能对响应时间的影响

然而，提高可扩展性的系统的响应曲线会向右移动，也就是说，在系统处于较高负载状态之前，曲线图中不会出现响应时间向不可接受的量急剧增加的那个拐点。如图 6-4 所示，性能不会改变（尽管这两者之间通常会有某种程度的耦合）。

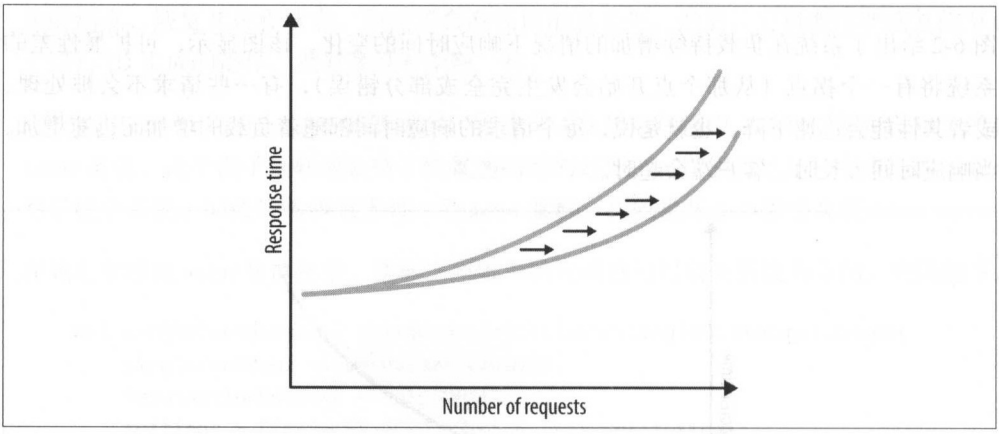


图6-4　可扩展性对请求数量的影响

系统的可扩展性倾向于沿着两个轴进行：水平扩展和垂直扩展。垂直可扩展性意味着如果在具有更多资源（CPU 能力、内存等）的较大机器上运行系统，则可以处理更高的负载。实现垂直扩展通常更容易，但不一定会很有效——一些系统并不会受益匪浅，因为它们一般还会受到除 CPU 和内存之外的其他资源的约束，因此增加垂直可扩展性不会让系统性能超出某一个临界点（这个临界点往往受其他因素影响）。

另一方面，水平扩展会更有意义。当增加整个系统的节点数量时（比如添加更多的子系统或节点），其实就相当于对系统进行了水平扩展。这样做可以使系统获得处理更多负载的能力。通常，这样扩展并不会导致任何性能点的增益：系统仍需要花费大约相同的时间去处理每个请求，只是有能力处理更多的请求了。

来看一个非常简单的计算问题，对两个整数求和，并考虑其可扩展性。

在单机系统上，添加更多的计算资源和内存可能会提高系统的性能，直到达到可以处理的网络请求数量的最大阈值。假定通常情况下，请求以某种方式通过网络进入系统。这时无论系统可以计算多少整数，处理速度如何，受限制的都是网络带宽，因为带宽会随着请求数量的增加而被耗尽，无法让更多的请求通过网络进入，所以系统就无法处理更多的请求了。

然而这个问题可以通过水平扩展很好地解决。通过添加第二台机器，系统可以处理大约两倍的请求量。在这个整数求和的简单示例中，正在构建的服务器集群中的所有机器都

会涉及请求需要经过的路由模块（例如网络交换机或负载均衡器），并且这类设备可以相当有效地进行扩展。

所谓的无共享架构是指，节点不会彼此共享任何信息，它们只是对整数求和，返回结果，并为下一个请求做好准备。

然而，随着对系统计算需求的增长，情况也会发生变化。一旦需要节点共享状态，无论怎样都可能引入导致系统瓶颈的问题。例如，我们需要计算系统总共处理了多少个请求，并通过某种 API 提供此数字。

计算集群的总请求数是一个惊人的复杂计算问题。这个数何时开始计算？是在请求结束还是开始时计算？如果要确保总数一直都是最新最准确的，就需要全局的状态，正如我们已经讨论过的那样，这是分布式系统中大多数问题的根源。只在单个节点上考虑计算请求总数其实是一件容易的事情（这样也会遇到相同的问题，只是变成了这类问题的缩小版），但是一旦添加了第二台机器，就会增加一个难度。

在分布式系统中，特别是在 Akka 中，遵循最终一致性以及采用 Actor 模型可以解决这类问题。与其实时计算所有机器的请求总数，不如让每个节点各自计算它们自己的请求数量，然后再汇总各个节点的请求数量来确定集群的请求总数。在 Actor 模型中，应该让每个节点上的每个请求都向该节点上的累加器 actor 发送一个消息来更新该节点的请求数，然后节点累加器会向集群单例累加器上报自己节点的请求数。最后，集群单例累加器就会获得该系统持续更新的请求总数的值，但会有一定的时间延迟。

处理计数实际上是一个很好的用于系统监控的例子，这是一种不同的计算风格，旨在对系统（在这种情况下是分布式系统）的操作进行实时可视化处理。

在高负载情况下，我们可能不会让 actor（例如对两个整数求和的 actor）针对每一个请求都上报一次计数。相反，actor 会持有请求的计数，并且累积 100 个请求才上报一次（除非短时间内没有新请求），这样来控制系统内的消息总量更易于管理。需要考虑到，允许触发上报的“累积”计数的数值会根据负载的变化而变化。这样，当负载低时，系统会在每个请求或每两个请求到达后就上报一次，但是在较高的数量下，系统只在累积了 100 个或 1000 个请求之后才上报。

正如我们所看到的，系统可以在短时间内变得非常复杂，但遵循 Actor 模型的模式一般会防止不利于系统提高可扩展性的行为发生。

可扩展性定义中的关键部分是“无故障”和“性能骤降”，所以要依次检查它们。

“无故障”意味着即使负载增加，系统也不会突然发生故障导致无法处理请求。随着资

源在集群上被全部消耗，系统应该以保持对外服务仍可用的方式“降低性能”。

这两者是紧密联系的，因为当 Actor 模型系统的部分功能失败时，其他部分不应该受到影响。这意味着从客户端的角度来看——客户端作为发送请求的组件——不应该有任何故障，即使一个或多个 actor 及节点确实存在问题。

如果有足够的节点（通常最小值应该是 3），系统的性能应该随着负载的增加而缓慢下降，因为一些节点（理想均匀分配）会变得饱和。

一个和可扩展性高度相关的概念是弹性。具有弹性是指具备在运行的系统中添加资源的能力，能够通过增加资源来扩展系统，而不影响客户端。

在这种情况下，如果可以向负载均衡器中添加新节点，并且这些新节点开始向集群单例累加器发送统计信息，则上述目标可以实现。随着集群的增长，我们可能会看到，可以向集群单例发送的消息量会成为限制因素，也就是系统的瓶颈，参考之前讨论的减少发送累积消息频率的策略可以缓解该瓶颈。

即使在这个比较简单的例子中，仍然可以发现监控是非常有帮助的：例如，如果看到正在计数的 actor 的未处理消息的队列长度随时间变得越来越长，则可以根据这些信息来调整累积因子，在系统中提供一些简单的自调整机制。如果使用这些信息来判断何时应该在负载增加时增加新节点，那么至少已经实现一半弹性了。另一半是相反的：当负载减少时，需要缩小集群。

如果计算资源是企业的成本，缩减集群则与扩大集群一样重要，弹性系统是节省成本的重要工具。

构建可扩展系统时通常围绕着几个基本原则，尽管这些原则的应用可能非常复杂。下面具体来看这些原则。

避免全局状态

如果可以避免全局状态，则可以绕过影响系统可扩展性的最大一个限制因素。如果需要这样的全局状态，可以封装一个集群单例 actor 来实现，但是这样做将牺牲系统的可扩展性。

避免共享状态

全局状态其实是共享状态的极端情况，它是集群所有节点之间的共享状态。如果最小化或理想化地完全消除这种共享状态，可扩展性的许多障碍也将消失，否则需要隔离 actor 后面的共享状态并仔细监控其影响。

遵循 Actor 模型

如果系统完全是用 Actor 模型构建的，而不是仅仅在某处使用 actor 而已，那么可以给系统提供适应负载变化的最大灵活性，因为系统的任何部分都可以利用 actor 的位置透明性进行进一步分发。

避免顺序操作

在前面的简单例子中没有遇到这种情况，但很多需要可扩展性的系统在操作时都有顺序要求。这是一种状态，也需要考虑时间因素，是一个双重风险。

如果某些操作必须等其他操作完成后才能执行，那么不得不让 actor 使用有限状态机（FSM）模型来应对这种情况，这在许多时候严重限制了系统的可扩展性。

虽然很难不依赖顺序而进行设计及发送消息，但可以通过一些谨慎的设计尽可能地避免依赖，并且从可扩展性的角度来说，这是值得的（更不用说系统本身的简单性也会增加了，这是令人愉快的副作用）。

隔离阻塞型操作

以上示例没有提到 I/O，但这是在任何基于 actor 的系统中引入阻塞型操作的常见原因。如果必须执行此类操作，请遵循 Actor 模型的规则，使用 Akka 把阻塞型操作的派发器和非阻塞操作的派发器隔离开，这样便可以单独优化它们了。将这些操作隔离到另一个线程池是保持应用程序高响应的关键。关于派发器，我们将在第 9 章中更详细地讨论。

监控和调优

在有负载的情况下监视正在运行的 actor 系统是正确衡量其行为的唯一方法，因为系统本质上是非确定性的，特别是在分布式的情况下。

要想知道监控什么以及它们会如何影响优化决策，请参见本书的其他章节。监控是一门值得研究的艺术，它可以帮助我们获得显著的可扩展性。

集群分片和一致性

在许多系统中，一致性和可扩展性一般是对立的。如果想要获得一致性，必须牺牲可扩展性。一致性要求集群中的节点之间共享信息，这降低了系统的可扩展性。但其实并不一定要这样，Akka 集群分片提供了一种弥合一致性和可扩展性之间鸿沟的方法，它允许创建平衡控制一致性和可扩展性的边界。但是，在深入研究如何做到这一点之前，首先

要理解什么是分片。

分片

分片已经在数据库系统中被应用了很长时间。它是一个强大的工具，可以扩展和分发数据库，同时保持数据库的一致性。数据库中的每个记录都附有一个分片键，此分片键用于确定数据的分布。不是将数据库的所有记录存储在集群中的单个节点上，而是通过分片键使所有记录分布在整个集群环境的节点上。

图 6-5 显示了一个非常简单的例子。这个例子中使用了数字键：所有奇数条目都将位于一个分片（节点）上，而所有偶数条目将位于另一个分片（节点）上。

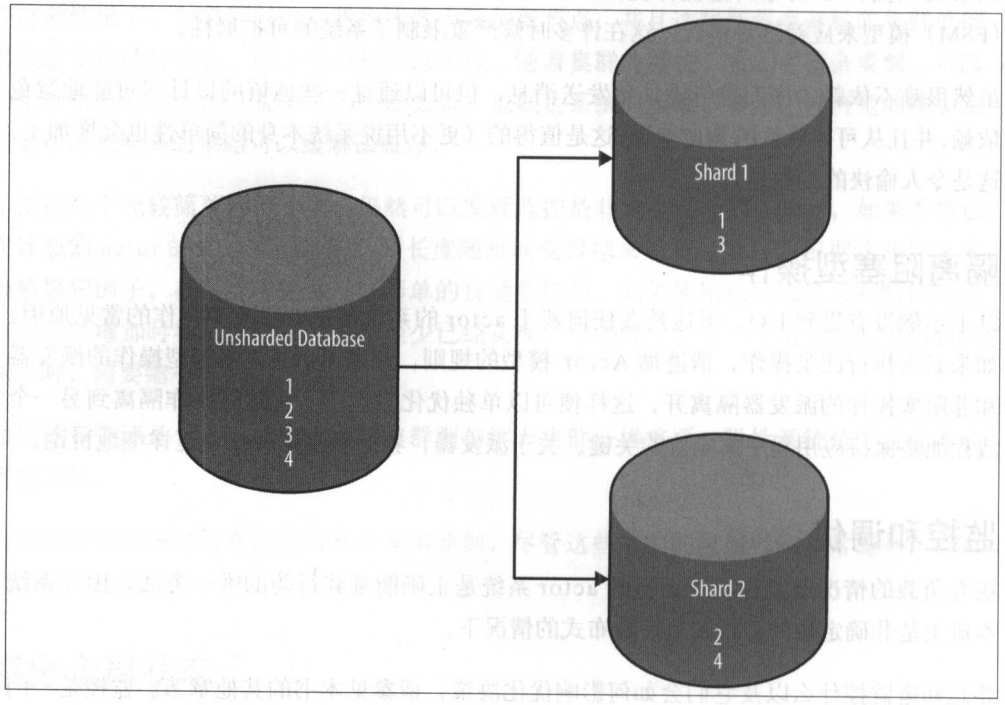


图6-5　对数据库进行分片的效果示意图

通过这种方式分发记录，竞争共享资源的情况可以减少，我们能够在集群中的节点之间均衡负载，而不需要把所有请求都转发到同一个节点。这样做也分散了一致性要求，对特定记录的所有请求将始终转到包含该记录的特定分片上。该记录的一致性只需要保持在该分片中即可，不需要保持在不同分片之间，因为它们不共享任何数据。

为了实现分片，需要用一个可靠的方法来确定特定数据所在的分片。通常使用特殊节点

来完成，它可以将流传递到适当的分片上。这个路由器节点不需要做很多工作，它只需跟踪分片，并根据需要传递流即可。虽然它仍然是系统的一个瓶颈，但它所做的工作是微不足道的，所以这个瓶颈的影响被最小化了。

Akka 中的分片

Akka 集群分片把这个分片思想进一步扩展了，它对整个集群中活跃的 actor 进行分片，而不是传统地对数据进行分片。每个 actor 都被分配了一个实体 ID，而且该 ID 在集群中是唯一的，它通常表示 actor 所建模代表的域实体的标识符。Akka 集群提供了一个从消息中提取实体 ID 的方法，还有一个函数用于接收消息并计算 actor 所驻留的分片的 ID。

发送消息时，使用上面提到的方法可以找到适当的分片，然后使用实体 ID 定位该分片中的 actor，如图 6-6 所示，这样可以在集群中找到该 actor 的唯一实例。如果目前不存在对应的 actor，则会创建 actor。分片系统将确保集群中每个 actor 都只存在一个实例。

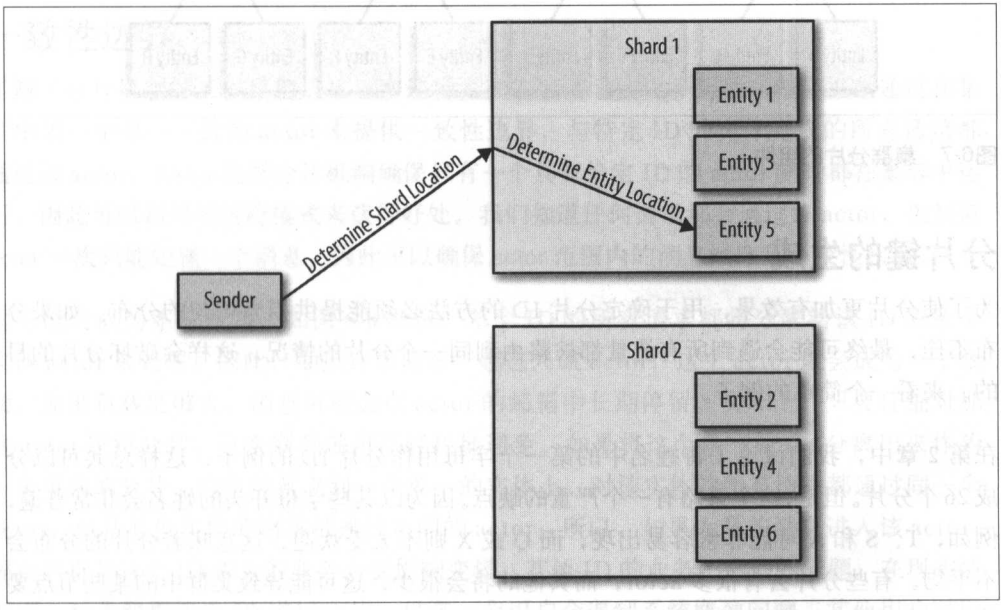


图6-6　分片系统中的消息流

集群中分布的分片所在的区域称为分片区域。这些区域作为分片的托管，参与分片的每个节点将为每种类型的分片 actor 承载一个单独的分片区域。每个区域可以托管多个分片。分片区域内的所有实体由相同类型的 actor 表示。

每个单独的分片可以托管多个实体，实体根据提供的分片 ID 的计算结果分布在不同分片上。

分片协调器（coordinator）用于管理分片所在的位置。协调器通知分片区域各个分片的位置。这些分片区域又可以用于向由分片托管的实体发送消息。该协调器是作为一个集群单例实现的，它在实际消息流中的交互被最小化了。只有在分片的位置未知的情况下，它才能参与这个信息流的交互。在这种情况下，分片区域可以与分片协调器进行通信以定位分片，然后该位置信息会被缓存，这样便可以直接发送消息给该分片，而无须与协调器进行通信来确定分片的位置。这里的缓存机制也意味着与协调器的通信应该是尽量少的，如果协调器成为系统的瓶颈，影响也是很小的。图 6-7 给出了集群分片的描述。

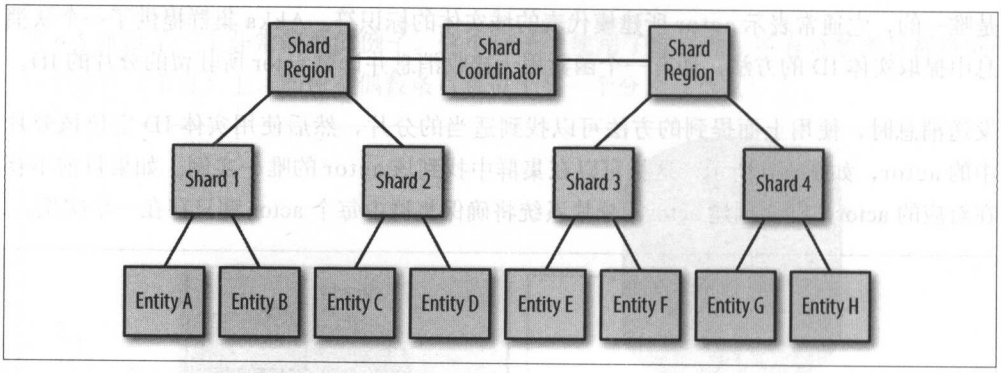


图6-7　集群分片的组件

分片键的生成

为了使分片更加有效果，用于确定分片 ID 的方法必须能提供相当均匀的分布。如果分布不佳，最终可能会遇到所有流量都被路由到同一个分片的情况，这样会破坏分片的目的。来看一个简单的例子。

在第 2 章中，我们讨论了将姓名中的第一个字母用作分片 ID 的例子，这样总共可以分成26个分片。但是，这个策略有一个严重的缺点。因为以某些字母开头的姓名会非常普遍，例如，T、S 和 A 可能非常容易出现，而 Q 或 X 则不太受欢迎。这意味着分片的分布会不平均。有些分片会有很多 actor，而其他的将会很少，这可能导致集群中的某些节点要接收大量请求，而其他节点会比较空闲。

我们的目的是使实体 actor 在所有分片上均匀分布。实现此目标的一个常见方法是用实体的唯一标识符去计算数字散列。这会产生一组相当随机的结果，应该是一个相对均匀的分布。然而仅仅这样并不够，因为只是简单计算一个数字散列将产生非常大量的分片，太多的分片会增加集群的维护成本，因此需要减少可能产生的分片的数量。可以根据想要的分片数量来对散列值进行取模运算，达到控制分片数量的目的，代码如下。

```
(entityId.hashCode() % maxShards).toString
```

分片的分布

该如何确定合适的分片数量呢？分片数量太多会对分片系统造成很大的维护负担，太少又会造成其他影响。

如果分片太少，可能无法将其均匀分布在集群中。例如，如果在由三个节点组成的集群中只有两个分片，则将有一个无效的节点，这显然不是我们想要的结果。相反，如果在三个节点的集群中有四个分片，集群中的一个节点将需要承担相当于其他节点两倍的工作量，这也是应该避免的情况。

一个好的做法是把分片数量设置为至少是集群节点数量最大值的 10 倍。这意味着集群中的每个节点将托管 10 个或更多个分片。这是一个很好的平衡，当从集群中添加或删除节点时，集群的重新平衡不会给任何单个节点造成重大负担。分片可以在集群中均匀分布。

一致性边界

了解了分片机制的工作原理后，下面来讨论如何使用它实现一致性？分片机制通过在集群中用一个单一一致的 actor 来提供一致性边界，与特定 ID 的实体完成的所有通信都通过该 actor。Akka 集群分片机制确保只有一个具有给定 ID 的 actor 随时都在集群中运行，因此可以利用单线程模式来获得好处。我们知道任何请求都会通过该 actor，也知道 actor 一次只能处理一个消息，因此可以确保 actor 范围内的消息顺序和一致性状态。

这可能会成为系统瓶颈。如图 6-8 所示，单个 ID 的所有请求都将分配给该 ID 的单个实体 actor 来处理。因此，如果许多请求一起进入该 actor，这个 actor 就会成为一个瓶颈。如果负载足够大，消息可能会在 actor 的邮箱中长期停留。实际上，只要仔细对那些 actor 进行分片，这个瓶颈就可以轻松地避免。如果将这个单一 actor 分离出来作为一个单独的分片，它将被孤立到一个单一的实体上。对该实体的所有请求都通过同一个 actor，但是其他实体的请求都通过不同的 actor。所以，如果大量的消息进入该 actor，那么只有与这个 ID 相关的业务会受影响变慢，其他 ID 的业务不会遇到问题。在现实情况下，如果根据用户 ID 进行分片，只有一个用户会遇到系统瓶颈问题，其他用户可以继续正常操作而不会遇到任何问题。

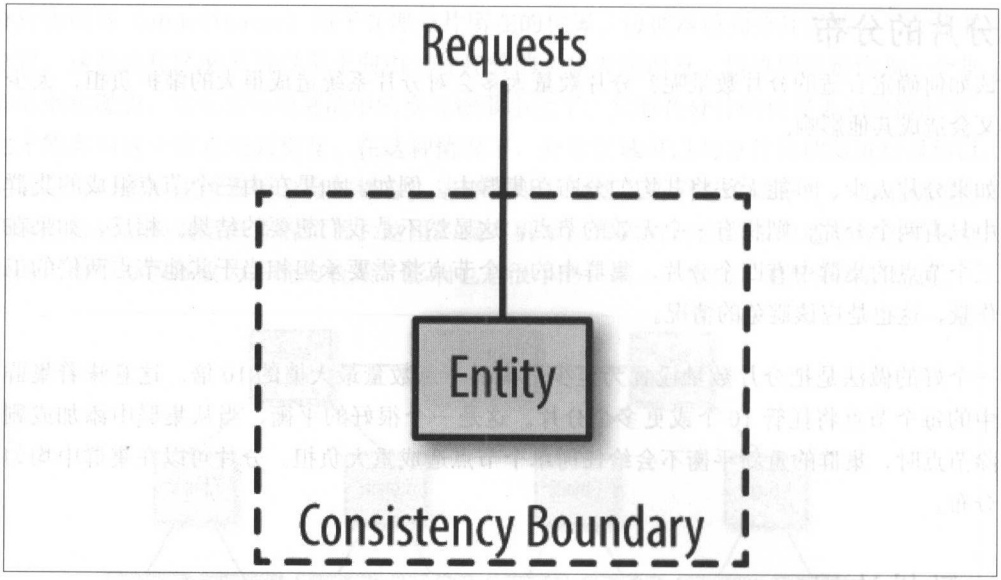


图6-8　实体的一致性边界

这种方法在 actor 的边界内为给定的实体提供了完全一致性，但是需要注意维持这种一致性的成本。根据 CAP 定理，只能从一致性、可用性和分区容错性三个特征中选择我们最需要的两个特征，永远都不可能同时拥有这三者。分区容错性通常是生产环境系统不愿意牺牲的特性，所以只能在一致性和可用性之间抉择了。在上面的例子中，我们选择了一致性，结果失去了可用性。如果托管分片的节点丢失或进行重新调整平衡的操作，那么在分片移动到另一个节点的过程中，实体 actor 会在一段时间变得不可用，即使移动很快，只需几秒钟。但是系统决定迁移的过程在很大程度上取决于系统配置的故障检测机制，根据如何设置故障检测，从发现故障、决定迁移、执行迁移、迁移完成，整个迁移过程可能会需要很长的时间。

可扩展性边界

我们使用分片来实现一致性；但是如果一致性和可扩展性是对立的，那么分片该如何提供帮助呢？围绕一致性创建边界，将其隔离为单个实体，可以使用该边界来实现可扩展性。一致性限于单个实体，因此可以将系统扩展到多个实体。

当需要扩展系统时，可以创建新节点并在这些新节点上重新分配分片，如图 6-9 所示。这样一来，以前由一个节点处理的工作现在可以被多个节点处理了，这样减少了单个节点需要执行的总工作量，产生了巨大的可扩展性潜力。

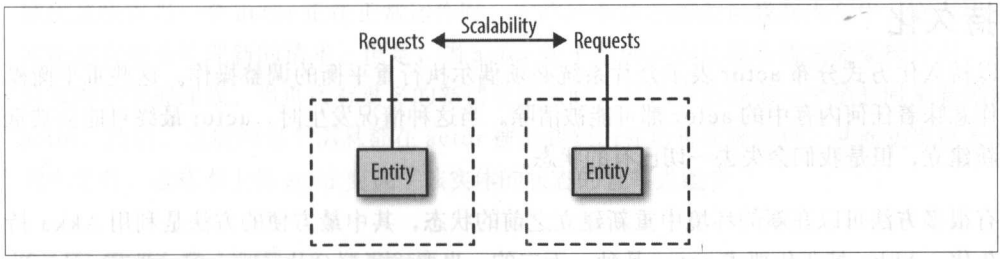


图6-9　缩放多个实体的同时保持这些实体内部的一致性

分片聚合根

分片机制很大程度上依赖于我们正确选择适合分片的实体的能力。首先要了解域中需要具备一致性的地方。那么该如何决定需要对其进行分片操作的合适候选者呢？

像往常一样，在决定应用分片机制的地方时，应该先回顾一下领域驱动设计（DDD）的原则。actor 必须是唯一可识别的，如果在域的基础上建模实现了 actor 系统，则 actor 将是域中的实体。接下来，我们可以更进一步。

在分片系统中，通常需要根据实体 ID 去找到对应的分片 actor，而且通常希望与该实体 ID 相关的所有操作都在本地机器上进行。也就是说，可以使用集群来定位已分片的 actor，但是在此之后还要最小化所有与远程通信操作相关的成本。最小化远程通信成本可以提高性能，因为这样可以消除网络延迟，也可以减少序列化和反序列化消息的操作。

通常，入手研究分片的好地方是系统的聚合根。聚合根提供了一个域内自然的一致性边界。我们通常关心该聚合中的数据是否一致，但是多个聚合之间的一致性又并不重要。在域中执行操作时，通常只能接触到单个聚合根，这意味着我们可以最小化网络通信。大部分操作可以由聚合根、其子节点或本地服务执行，只有在必要时才通过网络进行通信。减少网络通信量可以提高操作效率。

在项目管理域中，其中一个聚合根是人。这是研究分片的一个好的案例，从一致性的角度来看，我们可以保证所有消息都通过某一个特定的 actor 发给特定的人，不用担心发生将两个项目同时安排给同一个人的情况。单实体 actor 提供了阻止发生这种情况的一致性边界，这也很好地支持了可扩展性——可以在多个不同机器上处理多个不同的人，从而允许均衡负载。actor 系统的异步和分布式性质是我们所需的一致性和可扩展性的自然契合。

持久化

以持久化方式分布 actor 表示分片系统必须偶尔执行重平衡的调整操作。这些重平衡操作意味着任何内存中的 actor 都可能被清除。当这种情况发生时，actor 最终可能会被重新建立，但是我们会失去一切已有的状态。

有很多方法可以在新的环境中重新建立之前的状态，其中最方便的方法是利用 Akka 持久化。Akka 持久化要求 actor 是独一无二的，也要求集群分片是唯一的。这使 actor 的持久性 ID 很自然地成为实体 ID。

Akka 持久化为事件源提供了内置支持。发送给 actor 的命令将作为日志中的事件被持久化保存。当 actor 被重新创建时，可以重新触发被持久化的事件来重建以前的状态。这意味着 actor 可以随时从内存中被清除，因为使用事件日志随时可以为 actor 重建所有的状态。

当 actor 被持久化之后，可以确保当发生重平衡并且 actor 需要迁移到另一个节点上时，之前的所有状态都将被重建。因此该 actor 的位置不需要保持不变，它可以是非常灵活的，可以按当前需求改变。

钝化

期望一个系统始终在内存中保持所有的 actor 是不合理的。持久性的存在意味着我们不再需要让 actor 始终被加载在内存中，可以根据需要加载或卸载它们，并且因为它们是持久的，因此当它们被重新加载时，其状态将重新被建立。但是，如果是停止 actor，那么在该 actor 的邮箱中正在等待被处理的所有消息都将丢失。我们可以对 actor 的邮箱使用 `PoisonPill`，但是无法保证在使用 `PoisonPill` 后不再有其他新消息继续进入该邮箱。

为了缓解这个问题，分片机制中引入了钝化的概念。比起简单地停止 actor，钝化 actor 是更好的方法。这种情况下会给该分片区域发送一条消息，通知特定 actor 开始钝化，也会通知该分片区域开始缓存所有发给该 actor 的新消息。同时，actor 将发送一个自定义的消息，然后放置在它的邮箱中。最后，该 actor 会接收到这条自定义的消息，然后关闭自己。如果该分片区域中没有缓冲该特定 actor 的新消息，那么该 actor 将继续保持关闭状态。如果收到新消息，则会在该 actor 关闭后再重新创建 actor，并向其发送这些新消息。

通常这是通过在 actor 中使用 `setReceiveTimeout` 操作来实现的。此操作可以在给定的时间内没有接收到新消息的情况下时触发 `ReceiveTimeout` 消息。当等待时间结束后，actor 可以开始钝化。

钝化意味着当一个 actor 正在正常运作时，它的所有状态都会加载在内存中，它可以非常快速有效地处理新的请求，但是，当 actor 处于空闲状态且没有请求需要操作时，则会从内存中被卸载。当再次有更多的新请求进来时，可以在处理第一个消息时重新创建 actor，然后，之后的每个消息都在 actor 被加载到内存后再次被处理，从而实现巨大的效率提升。这基本上将 actor 变成了该实体的状态的直写式缓存。

使用集群分片保证一致性

只需要编写少量的代码就可以使用集群分片中最基本的功能了。没有持久化或钝化功能，任何 actor 都可以通过一些非常简单的改动来进行分片。主要的改动在于向 actor 发送消息的方式。我们需要将消息发送到分片区域，而不是直接发送给 actor。可以通过以下方式来创建分片区域。

```
val shardRegion: ActorRef = ClusterSharding(system).start(
    typeName = "People",
    entityProps = Person.props(),
    settings = ClusterShardingSettings(system),
    extractEntityId = extractEntityId,
    extractShardId = extractShardId)
```

此示例创建了一个名为 People 的分片区域。这个分片区域将托管 Person 类型的 actor。可以看到，我们通过 Props 构建了一个 Person，以便分片区域知道如何创建它们。该代码还传递了 extractEntityId 函数和 extractShardId 函数，可以接收传入的消息并解析它来提取必要的分片 ID、实体 ID 和实际消息。

在没有 Akka 持久性和钝化的情况下，这样做将会很有帮助，因为不需要更改其他信息。然而，由于在重平衡期间可能发生状态丢失的问题，因此非持久化分片的 actor 其实并不那么有用。那么，如果想要包含持久性和钝化，actor 应该是什么样子的呢？来看看下面的例子。

```
class Person extends PersistentActor {

  override def persistenceId: String = s"Person-${self.path.name}"

  context.setReceiveTimeout(timeout)

  override def receiveRecover: Receive = {
    case AddToProject(project) =>
      // Update State
  }

override def receiveCommand: Receive = {
```

```
      case AddToProject(project) =>
        persist(AddedToProject(project)) { addedToProject =>
          // Update State
        }
        case ReceiveTimeout =>
          context.parent ! Passivate(PoisonPill)
      }
    }
```

这是包含持久性和钝化的 actor 可能的状态。这个非常简单的包含持久化的 actor 与任何其他持久化性质的 actor 的唯一区别是，此处存在 `context.setReceiveTimeout (timeout)`。如果在给定的超时时间内没有收到任何新的消息，这个超时将使 actor 收到 ReceiveTimeout 消息。发生这种情况时，我们将 `Passivate (PoisonPill)` 消息发送给父级 actor，反过来指示托管该分片的分片区域使用 `PoisonPill` 关闭该 actor，也可以使用一个自定义的关机消息，该消息将被传递给 actor 来执行。

那么，`extractEntityId` 和 `extractShardId` 函数又是什么样子呢？这将根据具体实现而有所不同，先来看一个非常简单的例子。

使用集群分片时的常见做法是创建一个信封消息以包含分片所需的各种信息。此信封不是必需的。如果所有信息都存在于现有消息中，可以根据需要来设置提取器并提取信息。如果信息不存在，则可以将信息包装在信封消息中，代码如下。

```
case class Envelope(entityId: EntityId, message: Message)
```

信封信息可以采取最适合业务需求的任何形式。在这个例子中，信封信息只是将 `entityId` 和消息作为单独的部分来包含。

提取器的功能实现如下。

```
def extractEntityId: ExtractEntityId = {
  case Envelope(id, message) => (id.toString, message)
}

def extractShardId: ExtractShardId = {
  case Envelope(id, _) => (id.hashCode % maxShards).toString
}
```

`ExtractEntityId` 实际上是具有签名 `Partial Function[Any, (String, Any)]` 的偏函数的别名。这种类型的别名使代码具有了更好的可读性。可以看到这个函数只提取信封的元素并将 `id` 转换为字符串。

`ExtractShardId` 是具有签名 `PartialFunction[Any, String]` 的偏函数的别名。在这

种情况下，它可以计算分片 id，然后将数值转换为字符串。

需要强调的是，信封消息和提取器的功能可以按实际需求进行简单化或复杂化。如果用例需要额外的信息存储在信封消息中才能计算分片键，那么当然可以添加这些额外的信息。该信封消息或提取器的结构没有什么特别之处，因此可以随意定义。

尽管如此，一般来说，开始时最好尽量简单化。尽量使用一个非常简单的信封消息（或根本不使用），并在提取器中做最少的工作。随着应用程序的复杂性增加，如果有需要可以再来丰富它们的功能。

使用集群分片可以创建在一致性和可扩展性之间提供良好平衡的系统。它允许我们在一些明确定义的边界内相对容易地将系统扩展到多个节点。结合 Akka 持久化和钝化，可以防止在重新平衡或节点故障的情况下丢失状态或丢失信息。但是，我们仍然需要接受一个事实：尽管这些技术可以帮助处理某些类型的故障，但它们并不能涵盖所有的问题。在第 7 章中，我们将更详细地介绍应用程序可能遇到的不同类型的故障，以及使用 Akka 来缓解这些故障的方法。

结论

本章讨论了一致性和可扩展性之间的平衡，并给出了分布式系统中适当的一致性的例子，在后面章节中，我们将更加仔细地介绍故障情况，教大家当部分系统发生故障时应该如何应对？

高度分布式的系统比单节点系统更有可能发生部分功能失效的情况，因为分布式系统中含有更多可能会出错的子系统。

然而，正如我们将在第 7 章中看到的那样，一个正确构建的分布式系统因为故障导致整个系统完全失败的可能性实际上很低。

第 7 章

容错

开发人员具有一个共同的特点：追求完美。我们都会努力构建对故障有足够弹性的系统，通常不想承认故障是无法彻底避免的这一事实。现实是，无论我们如何努力追求完美，总会有一些超出控制的地方。即使可以控制，bug 也会进入系统。没有完美的系统。我们试图去预测每一个可能的故障点，但事实是，可能的故障点是无穷无尽的。即使可以建立完美的软件系统，也必须考虑到硬件故障。如果考虑了硬件故障，还必须考虑网络分区故障会如何影响系统。当发生飓风等自然灾害而破坏了数据中心时会发生什么？即使计划将软件和硬件的各个方面都设计得完美，但是控制之外的外部依赖还是会发生故障导致系统失败。

承认并接受无法避免故障问题发生这一事实或许会更好。与其试图处理每一种可能的故障情况，不如确保在意外发生时可以正确恢复，这将使我们处在一个更主动的位置。不必去试图预测未来从而建立一个完美的系统，而是应该建立一个足够聪明的系统来应对各种意外的状况。认识到失败是生活的一部分，我们应该去拥抱它，而不是试图忽视它。

处理故障在许多方面意味着要放弃全局一致性。试图保持全局一致性使我们走上忽略时间因素的道路，同时也意味着忽略故障。我们必须假设，当向某系统发送消息时，系统当前可能不可用，或者消息可能会丢失。在这种情况下，如何保证一致性？如果系统出现故障，唯一的选择是通知终端用户有关该错误的信息。这样避免了潜在的不一致问题，反过来却又产生了负面的用户体验。这是 CAP 定理中经典的决择问题：我们不能同时拥有一致性、可用性和分区容错性，因此必须要牺牲一致性或可用性。

与 Akka 一样，从现实世界的角度去思考系统往往是有帮助的，所以考虑一个现实生活中的例子：客户在餐厅点菜，我们会将客户视为终端用户，而餐厅的工作人员和他们使用的工具则是我们正在建立的软件。当客户进餐厅并点完菜时，意味着她已经发出了一

个请求，并且希望在一定时间内被及时处理。但如果这个过程中发生故障会怎样呢？

在传统的软件开发过程中，我们经常以下面两种方式之一来处理故障：抛出一个异常或在进行函数式编程时返回一个失败值。然后我们期望代码的调用者处理错误，或者通知用户该信息错误。这样做合理吗？

假设在客户点餐结束后，系统在与厨房沟通时发生了错误，导致厨房准备了错误的菜品。系统是否应该回到客户那里并让她来处理这个问题，告诉客户她的饭菜可能会被延迟。其实客户并不需要知道这些细节，而且绝对不要期待客户会为此错误采取任何行动。如果订单输入正确，但厨师在准备过程中弄错了食物怎么办？厨师是否应该期许服务器来处理该问题？服务器是否能够处理这个问题？在大多数情况下，答案将是“不”。如果我们在工作中犯了错误，应该竭尽全力去解决问题。如果由于某种原因导致失败，我们可以找到经理并寻求帮助，而不是去找客户并要求她来解决问题。

那么为什么构建的软件可以期待调用方来处理这个问题呢？无论是抛出异常还是返回故障，我们都是将错误返回给客户，让他们来负责处理，而不是将其推送至发生故障的地方。有些时候，要解决的问题已经超出了我们的能力，我们需要寻求客户的帮助。在餐厅例子中，如果客户点了一些今天已经卖光了的菜品，我们可以回去找客户，并让她选择其他的菜。但一般来说，这应该是最后的手段，只有用尽所有可以从失败中恢复的方法都无效时，才应该采取这种手段。

Akka 是基于不需要调用方负责处理故障的想法而建立的。Akka 主张由发生故障的 actor 负责处理问题，在 actor 不能处理的情况下，它会向其“监督者”寻求帮助。我们接受因为偶尔的系统故障导致的延迟问题，但这并不意味着客户的请求永远完成不了。因此要勇于面对系统的异步特性和可能发生错误的缺点。

故障类型

在详细讨论处理故障的不同方法之前，首先要认识到系统中可能发生的故障的不同类型。通常当我们开始讨论失败和故障时，往往会考虑到异常和崩溃。虽然这些是可能发生的常见错误，但是在开发响应式系统时还需要考虑其他类型的故障。

异常

异常是系统故障中最常见的类型之一。当某些代码出现故障时，它将作为异常事件以冒泡的形式被传递出来，一般包含有关异常性质的一些有用信息，包括消息和调用堆栈。可以利用这些重要的相关信息来调试错误并精确跟踪系统发生了什么。一个精心设计的

异常可能成为从错误中恢复系统的有用工具。

异常可能在许多不同的情况下发生。例如，调用第三方库中的代码（例如Java库）时可能会发生异常，或者只是数学运算的错误也会导致异常。在对Scala中的某些集合类型执行非法操作时（例如，对`None`调用`.get`）同样会发生异常。其中一些异常可以通过重新评估审视类型或代码来处理（例如，在`Option`上调用`.get`通常是发生某些错误的标志），例如代码中可能会发生的异常，但其他异常则可能发生在无法控制的情况下。我们无法控制外部第三方库是否会导致异常，这种情况下可以在自己的代码中用Try结构把异常捕获，而不是抛出它，但这并不会改变在某些时候必须处理异常的事实。

JVM中的致命错误

当然还有其他类型的错误异常，其中有一些是无法很好处理的。在JVM中，像内存不足这种严重的错误无论如何处理都会导致系统崩溃。捕获这种错误是毫无意义的，所以这种情况下把它包裹在Try结构中不会有任何帮助，Try无法处理这种会使系统崩溃的严重错误，因为它只能处理那些非毁灭性的故障。

像这种导致系统崩溃的故障都很难处理，已经在actor系统可以处理的范围之外了。虽然不能用正常的方法来处理它们，但这并不意味着可以忽略它们，反而要用更加聪明的办法处理它们。

外部服务故障

处理外部API会产生一系列的挑战。通常这些API通过使用某种形式的网络协议实现可以供外界使用的开放式接口，但是网络永远不会100%可靠，有时可能会遇到连接失败或其他类型的故障。此外，API在使用时通常会保持响应，甚至返回结果，但有时这些结果中会包含错误代码，这些都是可控范围之外的，很少有通过修改外部API以消除错误的可行选择，也绝对不可能防止所有可能的网络问题。这种情况下只需要像处理外部库那样处理这些错误即可，可以选择将错误包含在一个异常或Try中，但是无法完全避免。

不符合服务等级协议

即使建立了一个令人难以置信的可靠系统，仍然会存在故障需要处理。即使能够构建一个绝不受网络问题影响的系统，并且代码中根本不存在导致异常的条件，仍然可能会遇到故障。尽管系统可能会保证总是返回一个结果，而且该结果永远不会是一个异常结果，但是系统却不可能保证在合理的时间内返回结果。最好的系统仍然可能成为高负载情况下的受害者。在高负载情况下，这些系统可能无法达到预定义的服务等级协议（SLA）。

从技术角度来看，花费 10 秒的时间完成用户请求算是成功的，但从业务角度来看，它可能代表彻底的失败。如果该请求只是一个简单的网页加载，那用户很有可能等不了 10 秒就已经离开了。这些类型的错误故障其实挺有意思的，因为软件本身可能认为自己正在做“正确的”事情，但实际上却是在做没有考虑时间因素的错误的事情。

操作系统和硬件级故障

当然，即使创建了一个总是返回正确结果的软件，从来不需要与网络进行通信，性能完全可以满足所有的业务需求，并且永远不会用尽内存，这种情况依然可能遇到系统故障。当运行该应用程序的硬件出问题时会发生什么？如果硬盘驱动器发生故障或机器的操作系统崩溃会怎么样？这些都是无法控制的故障，但仍然需要处理。当无法完成用户请求的时候，和用户解释说因为硬盘驱动器失败导致问题，用户是否能接受解释？需要让系统停运多久才能修复这个问题？由于没有适当的预防措施，将会失去多少业务？

故障隔离

如我们所看到的，在软件系统中可能会出现许多不同类型的故障，甚至是一些很简单的故障问题。随着系统复杂性的增长，这些故障的潜在复杂性也在增加。如果构建系统时不够周密，即使很不重要的错误也可能会导致整个系统崩溃。一个未捕获的除以 0 的错误数学运算异常都可能影响到系统的上层，最终导致系统崩溃。系统的高负载请求可能会让所有的线程都高负荷运行，从而导致系统中所有的操作都停止。

无论采取何种形式，处理系统故障的第一个关键步骤都是隔离它们。我们构建的系统需要能把某个区域中的故障与其他区域隔离开，使发生的故障仅影响系统的一小部分而不是整个系统。在这方面，Akka 提供了一些不同的工具可以帮助我们。

舱壁模式

在 Akka 中隔离故障的最简单的方法之一就是使用一种叫作舱壁（bulkheading）的模式。舱壁这个术语来自航海世界，船舶的内部是使用一系列密封的水密段建造的，每个部分用舱壁分隔，如图 7-1 所示。如果船舶的任何一个部分被破坏，舱壁将防止水流入其他部分。船舶的一部分舱位将被水灌满，但其他地方的舱位仍然可以保持安全状态。这样允许船继续运行，避免了灾难性故障。

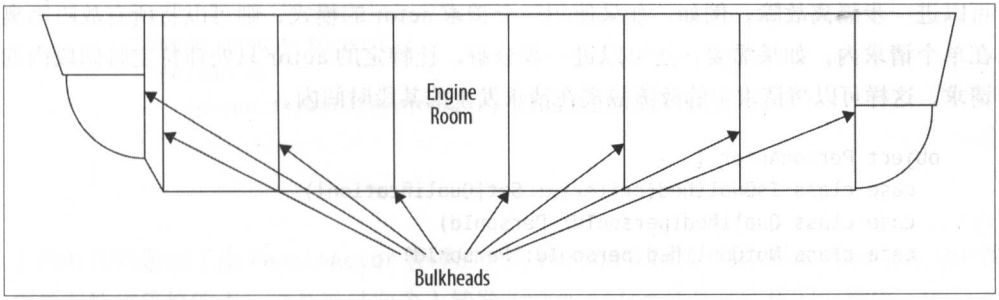


图7-1 舱壁

Akka 提供了以 actor 层次结构形式实现的舱壁模式工具。设计良好的 actor 系统可以使用这些层次结构在应用程序中创建隔离的故障区域。如果系统的某些部分发生故障，则该故障与包含故障的分支将被隔离。其他分支不会受到影响，可以独立于故障继续运行。

考虑以下的问题。假设我们正在尝试完成一个项目，有几个人的条件可以满足项目的具体需求。如果我们用 actor 来表示每个人，那么就可以向每个 actor 发送一个请求消息，并让它们根据要求的标准进行自我评估。由于使用的是 actor，因此可以同时进行评估（这是一个很大的优势），也可以在应用程序中创建故障区域，如图 7-2 所示。假设系统中有一些坏数据，而且有一个 actor 在处理请求时遇到这个坏数据然后发生了故障。如果这是唯一一个遇到坏数据的 actor，那么我们一般不希望这种故障影响到其他 actor。所以可以隔离发生故障的 actor，让其他 actor 继续处理请求，这样剩余的 actor 还能继续完成任务。所以这个过程中一定会产生一些有关错误数据的警报，但是不至于使整个请求失败。

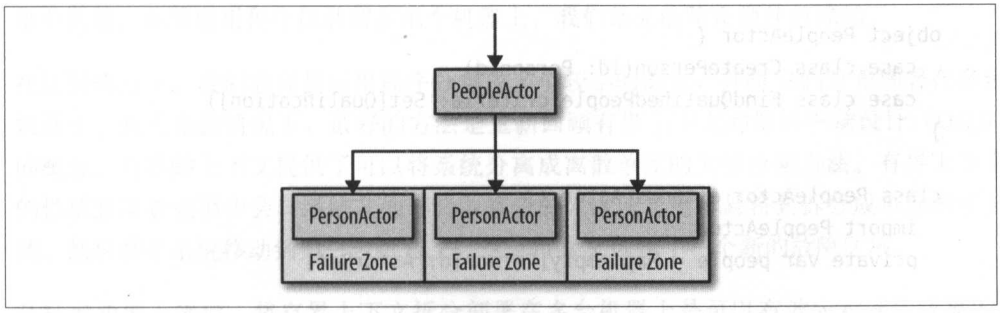


图7-2 两个有界上下文

这种隔离 actor 故障的方法不仅适用于坏数据，还适用于对外部 API 的失败调用，甚至是特定的 actor 负载过载等场景，遇到这些情况都应该把故障隔离在 actor 内部（尽管在某些情况下，actor 的派发器也可能会受到影响）。通过创建强大的 actor 层次结构，我们

可以进一步隔离故障。例如，如果使用一个请求 actor 的模式，则可以将所有故障隔离在单个请求内。如果需要，还可以进一步分解，让特定的 actor 只处理特定时间段内的请求。这样可以将请求中的故障隔离在请求发生的某段时间内。

```
object PersonActor {
  case class IsQualified(criteria: Set[Qualification])
  case class Qualified(personId: PersonId)
  case class NotQualified(personId: PersonId)

  def props(personId: PersonId): Props = {
    Props(new PersonActor(personId))
  }
}

class PersonActor(personId: PersonId) extends Actor {
  import PersonActor._

  private def isQualified(criteria: Set[Qualification]): Boolean = ...

  override def receive: Receive = {
    case IsQualified(criteria) =>
      if(isQualified(criteria)) {
        sender ! Qualified(personId)
      } else {
        sender ! NotQualified(personId)
      }
  }
}

object PeopleActor {
  case class CreatePerson(id: PersonId)
  case class FindQualifiedPeople(criteria: Set[Qualification])
}

class PeopleActor extends Actor {
  import PeopleActor._
  private var people = Map.empty[PersonId, ActorRef]

  protected def createPerson(personId: PersonId) = {
    context.actorOf(PersonActor.props(personId))
  }

  override def receive: Actor.Receive = {
    case CreatePerson(id) =>
      people = people + (id -> createPerson(id))
```

```
      case FindQualifiedPeople(criteria) =>
        people.values.foreach {
          person =>
            person.forward(PersonActor.IsQualified(criteria))
        }
    }
  }
```

上面的代码显示了由 PeopleActor 管理的一组 PersonActor，可以要求 PeopleActor 找到具有特定资格的人。这是通过向个人转发 IsQualified 消息来完成的。然后，actor 可以响应原始请求，表明它们是否能胜任。每个 PersonActor 都具有自己的故障区域，如果有 PersonActor 发生故障，它不会影响其他 PersonActor 继续工作，异常可以被隔离在特定 actor 的内部。

当然，这只适用于某些类型的故障。尽管舱壁模式可以有效地应对异常故障、SLA 故障和一些外部 API 故障，但在面对致命故障时不一定有效，处理硬件故障也无法完成。对于这些类型的故障，我们需要使用不同的方法。

优雅降级

仅使用一个 actor 层次结构来实现舱壁模式的问题在于，这种方法仍然限于单个机器和单个 Java 虚拟机（JVM）。如果硬件发生故障并且整个机器出现故障，那么整个系统都将崩溃。或者，即使没有硬件故障，但如果我们消耗完了机器上的所有可用资源，应用程序的响应仍然会变慢甚至无法进行。无论哪种情况，我们都会被用户抱怨。

为了真正防止故障，我们需要隔离与硬件相关的故障。但是要做到这一点不能仅仅考虑单个机器。如果应用程序仅驻留在单个机器上，我们是无法防止硬件故障的。

在这种情况下，我们需要将应用程序分解成多个较小的部分，并将它们分布部署在多台机器上。在大多数情况下，最好的方法是重新回顾有界上下文的领域驱动设计（DDD）的概念。有界的上下文提供了可以将系统分离成离散单元的天然分割方法。有界上下文的性质意味着它很少会与系统其他部分共享资源，这使它更容易将其拆分成单独的子系统，然后将子系统移动到另一台机器上。这反过来又创建了一个新的故障区域。

从技术角度上来说，将有界上下文拆分部署在多台机器上是可以有效应对硬件故障的。这样如果一台机器出现故障，我们只是会让这台机器上的子系统停止运行，但是运行在其他机器上的部分还可以继续运行。尽管出现了硬件故障，但是系统中那些不依赖故障子系统的部分仍然可以继续运行。

来看另一个例子。图 7-3 显示，示例系统是具有创建人员并记录人员技能信息功能的有界上下文。这个上下文称为技能矩阵。还有一个单独的上下文（即提案服务）将针对技能矩阵中的人员提出建议，根据人员的技能生成提案。如果这些都运行在单台机器上的单个 JVM 中，则一个故障就可能会导致整个系统发生问题从而导致整个系统停止运行。但是通过拆分这些有界上下文并在不同的硬件上部署运行它们，我们可以在一定程度上防止这样的致命故障出现。假设提案服务遇到与硬件相关的致命故障，因此系统失去了生成新提案的功能。但是，我们仍然可以创建新的人员并为这些人分配技能，因为系统的这部分功能是运行在另外一个单独的硬件上的。

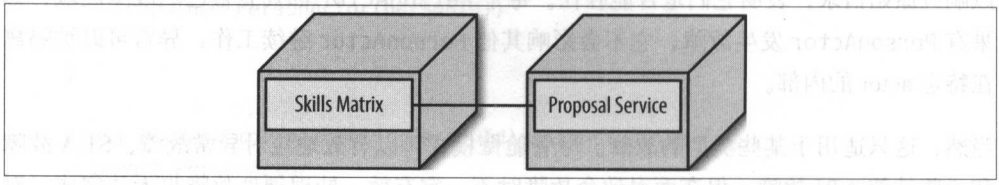


图7-3　远程故障区

另一方面，如果是运行“技能矩阵”的机器发生了致命的故障，而非运行提案服务的机器发生故障，这种情况下，故障类型便发生了变化。首先要看提案服务子系统是如何设计的，它可能依赖于“技能矩阵”子系统，如果是这样的设计，那么当“技能矩阵”子系统服务出现故障时，多少也会影响提案服务子系统，它将不再能够查找相关的人以及他们的技能，因为这是“技能矩阵”子系统的功能。但是这并不意味着整个系统都不可用，我们仍然可以查看以前的提案结果，这些结果都已经获取过，不需要再去依赖“技能矩阵”子系统查询信息。我们还可以使用提案服务子系统执行有限的操作，只要这些操作不涉及查找“技能矩阵”子系统即可。这里的重点是，虽然挂掉的“技能矩阵”子系统对提案服务子系统产生了影响，但是并不会对提案服务子系统产生严重的影响。

我们在日常使用软件时经常会看到这样的舱壁模式的例子：浏览一个网站，有没有注意到该网站的一小部分是不可用的？该网站的大多数功能可能还都是正常的，但其中的一部分已被禁用。我们可能会尝试在网站上执行某种操作但却得到如“对不起，此功能当前不可用”的消息。这种现象称为优雅降级。优雅降级可以避免使整个站点发生故障，我们只让其中的一部分不可用，而其他功能却不受故障影响，仍然保持可用的状态。

当然，这种舱壁模式对 Akka 来说并不是非常新鲜的。无论使用什么工具来构建应用程序，这都是一个很好的做法。那么 Akka 是如何结合它的呢？

首先，回顾一下全局实时一致性的概念。如果消除对全局实时一致性的需求，我们的系统就有了新的可能性。如果向无法立即响应的应用程序发送请求，则正常模式下，这个请求将会失败。但是，如果我们接受现实环境中的异步模式，也许这个请求就不会失败。

与其返回“对不起，此功能当前不可用”的消息，不如返回一条如“您的请求已被接受，处理完成后将通知您”的消息。如果只是系统的一个区域发生故障，并不意味着我们不能接受这个请求。只需要等待系统重新启动，然后再处理该消息即可。在某些情况下，甚至可能都不需要消息。如果要求 Facebook 发送一个朋友请求，那么无论是现在立即发送还是在接下来的三个小时内的某个时间才发送，这都不重要。如果请求比预期的时间要长很多，那么用户是否需要任何反馈？可能不会。当我们接受世界以异步的方式运作时，使可以开始建立一套支持这一运作方式的系统。以这种方式构建的系统自然更适合支持舱壁模式的隔离类型。

使用 Akka 集群隔离故障

舱壁模式是一种有用的技术，有许多方法可以实现它，其实它和是否使用 Akka 没有任何关系。Akka 的好处是，它内置了支持舱壁模式的功能，并使其变得更加透明。Akka 集群及其相关扩展提供了多种功能，以方便进行故障隔离操作。

通过使用 Akka 集群，我们可以透明地与远程 JVM 上的 actor 进行通信。虽然集群上的 actor 实例化方法略有不同，但实际的通信机制并没有变化。这意味着当我们决定将 actor 移动到集群时，不需要更改有关通信协议的任何内容。这样一来，拆分 actor 系统变得相当简单，我们可以将这些 actor 移动到不同的 JVM 或不同的机器上，只需要更改很少的代码即可实现。

当然，武断地决定这样做可能不是一个好主意。相反，当使用 Akka 集群来隔离故障时，也应该遵循特定的模式。如果 actor 层次结构中具有表示有界上下文的部分，那么就应该选择使用 Akka 集群来进行系统拆分。

也可以使用像 Akka 集群分片这样的工具来分割多台机器中的 actor 集合。使用集群分片可以把一组 actor 拆分成多个子集。这些子集可以驻留在不同的机器上。例如，我们可以对 Person actor 进行分片操作。好处是，如果某个“分片”发生故障，只有部分 Person actor 将受到影响，而系统中其余 actor 仍能继续运作，不受故障的影响。集群分片的另外一个好处是，它可以自动将失败的节点上的 actor 重新分配给其他节点。因此，如果丢失了一个分片节点，则可以在集群中的其他分片节点上重新创建这些 actor。当然，失败的节点正在进行的所有工作都将丢失，除非有恢复该工作的方法，但至少该 actor 的不可用状态只是暂时的。

使用熔断器控制故障

假设系统遇到导致其不再符合 SLA 协议的故障，使得对系统特定部分的调用都已经超时。

那么，我们可以使用舱壁模式来隔离该故障，以便系统的其余部分可以继续运行。也许这个失败是由系统特定区域的高负载导致的。

考虑一下，如果另一个请求进入系统的故障区域会发生什么。该请求会被加入到故障区域的队列里面，发生故障的系统将要处理它。但是该故障系统因为故障问题已经在努力处理先前未处理完的请求了，因此新增加的请求只会使事情变得更糟。随着更多的请求被继续发送到故障系统内，系统问题会加剧，请求需要等待越来越长的时间才能被处理，导致故障系统更加不可能赶上进度。同时，所有这些请求均没有达到 SLA 要求，这意味着它们也将超时。然后该故障系统甚至可能会重新处理该请求，希望第二次尝试可以成功，但这会使现有问题变得更加复杂。

下面来看一看 Akka 如何改善这种情况。Akka 提供了一种名为熔断器的工具，其在概念上与家中的电路熔断器相似。电路熔断器的目的是检测电气系统中的故障并关闭电流，防止问题变得更糟。Akka 熔断器具有相同的用途，它可以检测故障并关闭消息流，以防问题升级。

Akka 熔断器用来封装可能被认为是“危险”的调用。危险的调用可以是返回值或引发异常的同步调用，也可以是返回 future 的异步调用。在操作正常的情况下，调用将按预期处理。每次成功执行后，故障计数器重置为零，这时熔断器处于“闭合”状态。

当熔断器封装的危险调用开始发生故障时，熔断器的行为也会发生变化。故障计数器随着故障发生开始自增，每次增加 1 直到达到配置好的最大值。当达到这个最大值时，熔断器进入“打开”状态。在打开状态下，熔断器会自动使所有调用失败，甚至不会尝试处理该调用请求。相反，它会自动产生故障消息并返回给调用方，在这种情况下，故障消息是 `CircuitBreakerOpenException`。因为没有必要去处理请求，这些故障消息会迅速产生。熔断器将在接下来的一段时间内（一般都是配置好的）继续工作在打开状态下。

预先配置好的时间段过去之后，熔断器将进入“半开”状态。在半开状态下，进入熔断器的第一个调用将被处理，就好像它处于关闭状态一样。如果此调用成功，熔断器将进入关闭状态，系统恢复正常运行。如果第一个调用还是失败，熔断器将返回到打开状态，并且继续使所有的调用都失败。如果返回到打开状态，它将在接下来的一段时间内（一般都是配置好的）都保持打开状态，如图 7-4 所示。

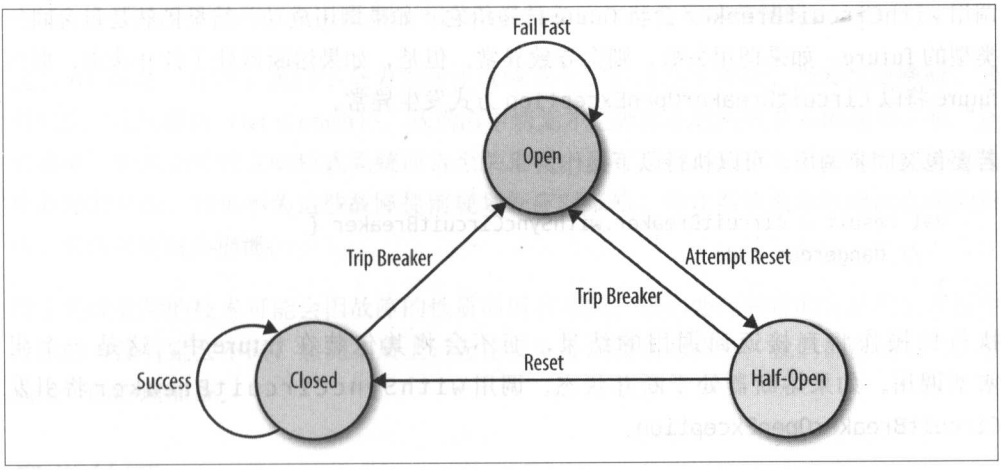


图7-4　熔断器的状态

使用熔断器的好处是，它提供了一个让发生故障的系统在一段时间内恢复原有功能的方法。与其让外界持续给它发送调用请求，使问题变得更糟，不如自动让这些调用失败，不再尝试调用发生故障的系统。我们应该在一段时间内持续拒绝请求，只有这样，系统才有机会自己恢复。系统恢复后，再调用就会成功，然后就可以继续正常运行。如果系统在设定好的时间段结束时还无法恢复，则在自动拒绝请求之前，我们只能向系统发送一次请求。通过这种方式，我们在该故障系统上只增加了很少量的负载，避免负载过度，这将有助于给系统提供一定时间去恢复原来的功能。

创建 Akka 熔断器相当简单，代码如下。

```
val circuitBreaker = new CircuitBreaker(
  context.system.scheduler,
  maxFailures = 5,
  callTimeout = 5.seconds,
  resetTimeout = 30.seconds)
```

上述代码中创建的熔断器最多允许发生五个故障。它还包括一个 `callTimeout`，以便将调用没有失败但完成时间过长的情况也算作故障。最后，有一个 `resetTimeout` 可以用来设置熔断器从打开状态转换到半开状态，系统必须等待的时长。

可以使用熔断器来包装同步和异步调用。对于异步调用，可以执行以下操作。

```
val result = circuitBreaker.withCircuitBreaker {
  (people ? CreatePerson(id)).mapTo[PersonCreated]
}
```

调用 withCircuitBreaker 会将 future 传递给它，如果调用成功，结果仍然是包含同一类型的 future，如果调用失败，则会导致异常。但是，如果熔断器处于打开状态，那么 future 将以 CircuitBreakerOpenException 方式发生异常。

若要包装同步调用，可以执行以下操作。

```
val result = circuitBreaker.withSyncCircuitBreaker {
  // dangerous code
}
```

执行该操作将直接返回调用的结果，而不会将其包装在 future 中。这是一个阻塞型调用。如果熔断器处于断开状态，调用 withSyncCircuitBreaker 将引发 CircuitBreakerOpenException。

故障处理

虽然故障隔离和故障控制对于构建强大的系统而言很重要，但这其实还不够。虽然我们可以使用优雅降级和熔断器等技术防止故障的蔓延，但是如果想从故障问题中恢复，该怎么办呢？或者是否可以简单地从一开始就防止故障的发生？

回到餐厅的例子。假设客户已经完成了下单操作，在为客户准备菜品的时候，菜盘突然掉到地上摔碎了。我们通过使摔碎的盘子只影响和它相关的菜的烹调进度成功地隔离了这个故障。服务器仍然可以继续执行新增的订单并完成其他的任务。餐厅可以继续接待新的客户。用完餐的客户离开餐厅后，清理他们的桌子，为接待下一桌客户做准备。换句话说，餐厅业务并不会因为这个故障而完全停止，因为这个故障只会影响与之相关的客户的订单。那么厨房方面又会如何呢？假设餐厅在某段时间内只有一个厨师，厨师除了丢弃受影响的订单还需要做一些其他的事情。只把故障隔离显然不够，在某个时刻，厨师必须重新处理这个订单，否则客户将会非常不开心。虽然餐厅里面还有其他用户在等着上菜，但这并不意味着我们就要完全丢弃发生故障的订单。所以除了隔离故障，我们还需要从故障中恢复它。该怎么办呢？我们显然需要重新处理订单，重新做菜。如果重新处理只需要很短的时间，那我们无须通知任何人，因为没有必要把故障细节告诉别人给别人添麻烦。如果需要一些时间来重新处理的话，我们可能就需要告知相关用户该故障的情况，但只是在有必要的时候。

想想刚刚发生的状况：订单被丢弃了，食物被糟蹋了。我们有试图阻止这个问题的发生吗？可能有关于餐厅运作的政策和程序有助于防止这些问题，但我们必须接受这些程序并不完善的事实。我们可以接受无论厨师多么小心，他都可能在某个时间点发出类似摔盘子的意外动作。这在一个繁忙的餐厅里是一件很正常的事情。因此，我们不要试图阻

止每一个订单发生故障，而是要制定相关的政策和程序来处理这些情况。

这与 Akka 是一样的：我们尽全力去防止故障发生，但这并不可能。当发生故障时，我们只能“让它崩溃（let it crash）”。从崩溃中恢复才是弹性系统应该具备的能力。有“让它崩溃”的意识对建立响应式系统而言至关重要。所有系统，无论复杂度如何，都有发生故障的可能。如果不为这些故障提前规划好应对策略，并在系统故障时做出合理的反应，那么系统就会崩溃。

用于处理故障的技术可能会因故障的性质而所有不同。我们处理异常的方法与处理硬件故障的方法肯定会不同，然而本质上，无论发生什么级别的故障，我们都应该有“让它崩溃”的心理准备。

异常处理

另一种常见的故障是代码自身引发的异常。本节将探讨处理这类故障的技术。

非致命异常

在一个应用程序中，我们需要关心的主要故障形式就是程序异常。更具体地说，需要关心非致命异常。这种异常不一定会使整个系统崩溃，我们可以在应用范围内从故障中恢复。那么该怎么做呢？

在单个 actor 中，可以处理可预测的故障。如果预料到某个方法可能会引发异常，就将该方法包装在 try-catch 块中，然后在发生异常时及时处理，这样可以把该异常隔离在这个 actor 内。但这并不总是有效的解决方案，因为有时 actor 内发生的异常可能会超出该 actor 处理异常的能力范围。

回到之前餐厅的例子，如果厨师摔碎了做好的菜，他可以重新再做一份。但是如果这道菜涉及的原材料都用完了怎么办？这时他就无法靠自己解决这个问题了，必须要靠其他人的帮助才能解决。厨师可能需要去找餐厅经理来确定最佳的解决方案。

在 Akka 中，当一个异常不能被它的 actor 处理时，该 actor 上层的监管 actor 就会被告知有这样的异常的存在，然后决定如何处理。一般来说，监管 actor 可以根据配置好的策略决定方案。每个 actor 都有能力创造子 actor，因此它就成为了监管 actor。所以每一个 actor 都有一个类似的策略，如果没有自定义策略，则会使用默认策略。

Akka 内置了两种类型的 `supervisorStrategy`：`OneForOneStrategy` 和 `AllForOneStrategy`。`OneForOneStrategy` 指示监管 actor 将恢复逻辑应用于发生故障的 actor，其他 actor 不受影响。相反，`AllForOneStrategy` 告诉监管 actor 将恢复逻辑

应用于所有的子 actor。

考虑一下调度域的例子。假设我们正在尝试安排一个项目，制定好该项目方案后，我们给多个 Person actor 发送请求查看他们之中是否有可以满足项目要求的合适人选。如果有一个 actor 在访问数据库时发生了故障，可能不会影响所有的 Person actor，而只有那一个 actor 受到影响。这是 OneForOneStrategy 的一个很好的用例。另一方面，如果发现项目内部的数据在某种程度上都是无效的，那么任何尝试处理该数据的 actor 都可能因为同样的原因发生故障。在这种情况下，如果等待所有 actor 发生故障就完全没有意义了，因为我们早已经知道会发生故障。这是 AllForOneStrategy 的一个很好的用例，这里的逻辑适用于所有的子 actor。

监管者的策略还包括一个 Decider。Decider 是一个 Partial Funaion[Throwable, Directive]。Decider 根据发生的异常类型来决定采取何种应对方式。Decider 基本上会将异常类型映射到以下指令之一。

Resume

Resume 指令告诉监管者发生故障的 actor 应该继续处理消息，就好像没有发生故障一样，状态不会发生变化。对于无状态的 actor 或者状态没有被故障影响导致失效的 actor 是很有用的。

Restart

Restart 指令告诉监管者发生故障的 actor 应该被重启。这意味着现有的 actor 将被停止，一个新的 actor 将代替它。这将使该 actor 的所有状态都被重置为初始状态。如果故障 actor 的状态已经因为故障而失效了，这是很有用的。

Stop

Stop 指令告诉监管者故障的 actor 应该被停止而不是被替换。如果故障导致该 actor 不再被系统需要或无法以任何方式恢复，该指令将非常有用。例如，如果 actor 是仅为完成当前任务而被创建的一次性的 actor，并且该任务失败，不再需要该 actor，那么便可以停止它。

Escalate

如果监管者无法为发生的故障提供适当的解决措施，我们就应该升级该故障的等级。因为监管者已经无法处理该故障了，因此应该由它的监管者来处理这个故障。这就像是重新抛出异常一样。

现在可以为其中一个 actor 创建一个 supervisorStrategy。People actor 有责任去监督 Person actor，这个 People actor 的简单功能如下。

```
override val supervisorStrategy =
  OneForOneStrategy() {
    case _: AskTimeoutException => Resume
    case _: IllegalArgumentException => Stop
  }
```

这是一个非常简单的例子。在这种情况下，如果收到一个 AskTimeoutException，可以假设 actor 的状态没有被破坏。可以尝试再次发送 Ask 请求或继续处理下一个消息，所以发生这种故障时我们只要简单地继续运行即可。对于 IllegalArgumentException，假设发生这种情况是由于传递到 actor 的构造函数中的数据有问题。如果这些数据在某种程度上是非法的，那么恢复或重新启动 actor 就不会起到任何作用，所以应该停止该 actor。

通常简单地提供这样的应对方案是不够的。假设我们通常都希望 actor 可以发挥一些作用，所以当发生故障时，我们不会简单地忽略它，而是会采取一些额外的恢复操作，比如重新发送那条失败的消息再尝试一次，也可能会把失败的消息重定向到另一个 actor 上，看看该 actor 是否可以完成任务，或者，也可能想要执行其他复杂的操作以便能从故障中正确恢复。

在餐厅的例子中，如果经理只是告诉厨师让他继续工作，这样的处理方法显然是不明确的。丢失的菜单需要重新生成，如果不能，则需要通过其他方式来恢复这个菜单，比如回到客户那边解释现在的情况。这些操作都需要由经理来执行。

监管层的策略不一定是一个从异常到指令的简单映射，也可以添加额外的逻辑。例如，当收到特定的异常时，可以向 actor 发送消息来处理该异常。如果需要，也可以更改监管者的状态。这些选择都为故障恢复提供了有意思的可能性，只要在异常中包含了足够的信息，监管者就可以执行从错误中恢复原功能所需的任何步骤。

来看一个很简单的例子。如果某条消息由于某种原因而无法处理，则可能会在异常中包含该消息，代码如下。

```
case class MessageException(msg: Message, cause: Throwable) extends
  Exception(s"The Message Failed to Process: $msg", cause)
```

当该消息可用时，可以在 supervisorStrategy 中使用该消息。

```
override val supervisorStrategy =
  OneForOneStrategy() {
    case MessageException(msg) =>
      sender ! msg
      Resume
  }
```

这是一个非常简单的自愈系统的例子。这种情况下，当出现异常时，可以将其包装在另一个异常中，并将消息包含在该异常中。这使得监管者可以访问该消息，接收异常后提取消息，将消息重发给 actor，然后指示 actor 进行恢复处理。这样便可以重发失败的消息。

事实上，这不是解决这个特定问题的最好办法，我们在这里只是将它作为一个简单的示例，告诉大家可以做的选择有很多，不仅仅是简单地返回一条指令。更好的解决方案是利用第 6 章中介绍的 `AtLeatOnceDelivery` 机制。这样可以更加灵活地自定义，同时重试操作也可以被持久化。

可以按业务需求简单化或复杂化这里的逻辑。在复杂的情况下，我们可能不会将消息发送给同一个 actor，因为这个 actor 已经发生故障，它已经不再有用了。相反，我们可能会将消息发送给不同的子 actor，或者专门为了处理异常而生成的一个新的 actor，也可能将消息添加到某种错误队列中，以便稍后处理。关键是我们认识到故障已经发生了，需要采取适当步骤从中恢复，以一种能自我修复的方式去构建系统。

当然，所有这一切都会引发一个明显的问题：我们只处理了已知的异常。如果发生未知的异常会怎么样？对于未知的异常，最好的处理方法是在 `Decider` 中设置一个可以处理非致命异常的策略，然后就可以处理这类异常了。乍一看，这似乎是可行的。但是如果我们事先不知道将发生什么异常，那么怎么知道如何正确地处理它呢？在一个非常具体的使用场景下，若 actor 设计良好且包含简单的消息协议，这种情况实际上是比较容易处理的。一个设计良好的 actor 只有几个可行的恢复策略。然后，只需要从非常有限的策略中选择一个来确定发生未知异常时的应对措施即可。如果发现自己有太多策略可以选择，而且没有一个是特别显而易见的好方法，很可能是因为 actor 太大了，这时则需要分解它。另一方面，如果 actor 已经相当简单，但仍无法确定如何处理某些未知异常，那么我们所处的 actor 的结构层次可能是错误的，这时需要向下移动到其中一个子节点来处理异常或升级异常级别让上层节点来处理。

JVM 中的致命错误

有时候，无论多么小心都会发生无法处理的错误。在这种情况下，无论是内存不足、堆栈溢出还是其他错误，系统都将中止。这种情况下，我们能采取的处理方式很少，基本不可能去修复 JVM 中的错误。系统崩溃，我们也随之崩溃了。

这是否意味着我们应该接受这类致命故障，并借助人工干预来恢复它呢？还是有工具即使在这种灾难性故障下也可以实现恢复？即使系统发生致命故障，我们也可以在系统设计阶段就采取预防措施来帮助它从最坏的故障中恢复。

为了让彻底崩溃的系统能自我修复，需要做的第一件事是提供一些让故障应用程序可以

重新启动的方式。Akka 自身没有现成的工具可以做到这一点，但是有一些其他工具可以借助。Lightbend 为此提供了解决方案：使用 ConductR。ConductR 是一种可用于创建集群并指定要运行的应用程序的数量的工具。在某个应用程序发生故障崩溃的情况下，ConductR 将确保该应用程序的新实例在集群中的某个地方启动来替代原程序。其他工具（如 Mesos 和 Docker Swarm 等）也提供了确保应用程序实例正常运行的方案。

无论使用 ConductR 还是其他应用程序，基本思想都是一样的。我们需要让一个应用程序以某种方式监视进程。如果进程失败，负责监控的应用程序将自动重启发生故障的进程。这有点像 Akka 监管者，只有在这种情况下，该指令才总是 `Restart`，而不是检测异常（需要通过使用其他方法来确定故障）。

监视应用程序状态并自动重启它们，从运维操作的角度来看可是帮了大忙。假设已经发现一个应用程序有严重的错误，导致它每隔几分钟便重启一次，但它依然能够为客户提供全天候的服务，因为它每次失败都会被重新启动。并且因为系统的异步响应式设计，客户端其实并不知道发生了故障。这种类型的设置可以让我们避免在凌晨 3 点接到可怕的系统报警电话。

仅仅重启应用程序明显还不够。这仅仅是朝着正确方向迈出的第一步，如同非致命异常错误一样，我们还需要采取某种自我修复机制来恢复故障。那么如何在系统中实现自我修复呢？

修复这些系统的关键是在应用程序中设置一个回退点。系统需要一个时间点，就像在沙子中画一条线，当系统运行到该时间点，系统数据就会是安全的。在此时间点之前，任何故障都将导致数据丢失；在此之后，任何故障都可以使用这个时间点对应的系统数据进行恢复。最理想的情况是将这一点设置在尽可能接近进程开始的地方。我们希望尽快达到系统安全时间点，那么实际结果究竟如何呢？

来看看调度域示例。在向调度域发出请求以执行特定项目的调度操作时，什么时候可以认为该请求已经执行完整了呢？当项目完成调度并且所有结果都已经计算完毕时，我们才认为它是完整的吗？那是最终的目标，但这个过程也许需要持续几个小时。我们不希望任何进程耗费几个小时的时间，对此又该如何简化呢？

处理这个问题的一个好方法是将这个命令视为在被接受和持久化之后就立即被处理完毕的。换句话说，当我们发出一个安排项目的请求时，系统接受这个请求，并把这个请求推送到一个持久化的队列中，然后我们可以向客户端发送一个确认消息，告诉客户端它的请求命令已经被接受并将被异步处理。如果在命令请求被持久化之前发生故障，那么客户端会接收到一个错误的回馈消息；如果在命令被持久化之后发生故障错误，系统将

重试或采取某种自我修复的逻辑。

有很多方法可以实现这种逻辑，但它们都依赖于一个简单的概念：至少交付一次。即使发生故障，也要保证消息能够被成功传递。一种方法是使用外部持久化的消息总线。我们需要一个消息总线，将消息传递给客户端，并允许客户端确认收到该消息。这样的总线可以提供自动重试的机制。当命令进入系统时，它执行该命令所需的验证，然后将其转换为另一个消息，再将该消息发送到消息总线。在将消息发送到消息总线之后，该请求被认为是完整的。此后，任何进一步的处理都将异步发生，故障不再会影响到它。该消息可能会由于故障错误而被延迟处理，但最终系统恢复后它依然会被处理。如果消息无法传递或未被确认，消息总线将在稍后重试。

在 Akka 背景下，我们可以使用像 Akka Persistence 这样的工具来实现类似的效果。另外，还有许多其他方法也可以做到这一点，先来看其中一种。Akka Persistence 中引入了“至少交付一次”的概念，可以利用此功能提供消息交付保证。Command 处理器接收命令，验证它，然后使用 `AtLeastOnceDelivery` 机制将生成的消息发送到系统中。如果系统无法接收并确认消息，Command 处理器将重新发送该消息，直到收到确认消息为止。这给我们提供了一种可靠的交付形式，也提供了正在寻找的重试机制。现在可以保证消息最终都将被处理。即使 Command 处理器可能会发生故障，但因为它是一个持久的 actor，因此当它被重新创建时，它将继续发送消息。

这就是发生系统故障时设置系统回滚点的作用。如果系统发生错误，并且无法成功处理该消息，则该消息将不会被确认，这将迫使系统稍后重新发送该消息。根据消息的复杂性，可能需要在消息处理流水线中创建多个回滚点。在整个流程中，可以在需要时使用 `PersistentActor` 和 `AtLeastOnceDelivery`，以确保系统遇到灾难性故障时可以重新创建必要的状态并继续。

以上方法只适用于采用异步设计的系统。如果系统的所有机制都是同步设计的，那么该系统默认不具备这种恢复能力。同步系统中不允许发生故障失败，系统确认命令完成之前，所有事情都必须成功完成。如果系统的部分模块出现了故障，则需要直接令该命令失败，不能花费 15 分钟去等得该系统的部分模块恢复。另一方面，如果系统的大多模块是异步的，这不是问题。系统可以接受请求并将其持久化。如果由于系统出现故障导致无法成功处理该请求消息，系统可以在故障恢复后再尝试处理。没有必要让使用者知道系统曾经发生过的故障。

外部服务的故障处理

大多数系统最终都要与外部服务进行通信。这种外部服务一般是在我们的控制范围之外

的。我们不能采取任何措施来提高外部服务的弹性，而且有时候外部服务会发生故障。那么该如何处理这些故障呢？

处理外部服务故障的技术实际上是处理内部故障技术的组合。需要做的第一件事是隔离故障，以便能够正常地对服务进行优雅降级。隔离这些故障的技术与处理内部系统的技术没有什么不同。可以将外部服务系统视为另一个有界的上下文，为它创建一个包装器来包装它。如果外部系统出现故障，执行特定操作的功能将受到影响，但是系统的其他部分仍然可以正常运行，例如那些不需要和外部服务系统进行通信的部分。

还可以使用持久化队列和其他技术，以便在外部系统发生故障不可用时通过它们来缓冲消息。当外部系统重新可用时，我们可以从持久化队列中取出消息，然后重新发送该消息，通过简单使用 `AtLeastOnceDelivery` 机制的包装器 actor 去包装外部服务系统可以实现。

来看一个简单的例子。

```
object ProjectAPIActor {

  case class CreateProject(projectId: ProjectId)
  case class ProjectCreatedSent(projectId: ProjectId)

  def props(remotePath: ActorPath):Props =
    Props(new ProjectAPIActor(remotePath))
}

class ProjectAPIActor(remotePath: ActorPath)
  extends PersistentActor
  with AtLeastOnceDelivery {
  import ProjectAPIActor._

  override val persistenceId: String = self.path.name

  private val projectActor = createProjectActor()
  protected def createProjectActor() = {
    context.actorSelection(remotePath)
  }

private def handleSend(msg: ProjectCreatedSent) = {
  deliver(projectActor) {
    deliveryId =>
      ProjectActor.CreateProject(deliveryId, msg.projectId)
  }
}
```

```
  private def handleConfirm(confirm: ProjectActor.ProjectCreated): Unit = {
    confirmDelivery(confirm.deliveryId)
  }

  override def receiveRecover: Receive = {
    case ProjectCreatedSent(projectId) =>
      handleSend(ProjectCreatedSent(projectId))
    case confirm: ProjectActor.ProjectCreated =>
      handleConfirm(confirm)
  }

  override def receiveCommand: Receive = {
    case CreateProject(projectId) =>
      persist(ProjectCreatedSent(projectId))(handleSend)
      case confirm: ProjectActor.ProjectCreated =>
        persist(confirm)(handleConfirm)
    }
  }
```

该代码提供了一个与 ProjectActor 通信的 ProjectAPI。ProjectActor 是远程的，可能会由于某种原因而发生故障。不过值得注意的是，无论 ProjectActor 是本地的还是远程的，都可以应用于这种技术。

远程的 ProjectActor 将负责处理命令消息。假设调用成功，它会将响应发送回发送方。有意思的是，消息协议中包含了 deliveryId，它是由 AtLeastOnceDelivery 机制提供的 ID。

在 API 中，发送消息时使用了 AtLeastOnceDelivery 机制。这里使用 deliver 方法，而不是通常使用的 tell 或 ask。该方法与 persist 调用相结合时，如果发生故障，系统可以尝试重新发起调用请求，另外还可以查看响应消息，当收到响应时，可以通过使用 deliveryId 来实现 confirmDelivery。使用 deliver 和 cnfirmDelivery 的组合，我们可以获取所需的交付保证。如果外部服务失败，则不会确认该消息。然后，消息将在可配置的延迟时间结束之后被重新传递。

如果需要，也可以在管道中的多个地方使用 AtLeastOnceDelivery 机制。此示例仅在外部服务中使用，但是并不表示不能在其他地方使用。请注意：过度使用该机制表明我们仍然在尝试实现全局一致性。AtLeastOnceDelivery 机制最适合用于调用流水线的两端，因此最好不要在管道的每个阶段都使用它。

结论

在本章中，大家了解了基于 actor 的系统如何将故障视为预期的状况进行处理，以及将每个级别的故障响应操作集成到正常操作中的方法。基于 actor 的分布式系统发生系统级故障的可能性非常低，同时比其他类型的系统更容易从故障中恢复。

Akka 有助于减少系统的停机时间。但实际情况是，即使提供故障隔离及从故障中恢复系统的方法，故障仍然是存在的。我们需要增加系统冗余，以便在发生故障时可以有让系统进行回滚操作的数据。在第 8 章中，我们将深入讨论在面临故障时可用于提高系统可用性的技术。

第 8 章

可用性

响应式编程的核心要求之一是，系统必须具有可用性。可是系统可用的意思是什么呢？如果系统能够及时响应一个请求，我们认为该系统是可用的。如果系统的响应时间过长或者根本不响应，我们则认为该系统是不可用的。系统在停机或超载时是不可用的。

可用性不同于系统扩展的能力（例如可扩展性）。

如第 7 章所述，有很多原因可能导致系统不可用，而且无论如何努力构建系统，都可能会在某些时刻因为某种原因导致系统的部分功能不可用。保持系统可用的关键在于，即使面对故障也可以找到缓解这种故障的方法。我们已经讨论过如何解决特定类型的故障，本章将从更高层面讨论如何以支持可用性的方式构建系统。

微服务和单体式应用

目前我们经常讨论的构建应用程序的方法主要有两种。一种是采用单体式架构，另外一种是采用微服务架构，但大部分应用都是处于单体式架构和微服务架构之间的某个中间状态，很少有应用程序是完全的单体式架构或完全的微服务架构。

单体式应用程序是指把所有组件都部署为单个单元的应用程序。单体式应用程序通过创建各种独立的库来隔离复杂性，然后把这些库编织在一起构建成一个完整的大系统。

另一方面，微服务是通过将较大的应用程序分成较小的服务来构建的。这些微服务以隔离的方式执行非常细微的任务。复杂性被隔离在各个独立的微服务之内，然后通过其他微服务进行组合。

可用性并不是单体式架构服务或微服务的问题。我们可以使用任何方案创建高可用性应用程序。但是，实现方式以及应用程序的可扩展性取决于选择的具体方案。

单体式应用程序中只有一个可部署的单元。在这种情况下，我们提供可用性和可扩展性的方式是把这个可部署的单元简单地复制为多份。如果某一个单元发生故障，其他单元将会帮其收拾烂摊子。这种情况下，Akka并不会提供帮助。部署的单元彼此间不需要通信，所以不需要引入像 Akka 集群这样的功能。在单体式应用程序中，我们可以使用 Akka 更好地利用资源，也可以使用它的舱壁模式和其他技术来隔离故障。Akka 确实有适合用于单体式应用程序的技术，但是当我们谈到 Akka 的可用性时，反而会对那些不适用于单体式应用程序的技术更感兴趣。

微服务提供可用性的方法和单体式应用程序差不多，它通过对每个微服务部署多个实例来达到这个目的。但是，在这种情况下，由于每个微服务是分开的，因此可以独立部署，独立扩展，独立调整可用性。系统中某些至关重要的地方可能要求高度可用。其他地方可能不太重要，每天只需要运行几个小时甚至更少，这些地方对可用性的要求要低得多。

当多个微服务需要彼此沟通时，Akka 提供了帮助工具。但是，在详细介绍 Akka 提供的工具之前，我们要先讨论如何将应用程序分解为微服务。

用有界上下文划分微服务

领域驱动设计（DDD）为我们提供了一种非常自然的将应用程序划分成较小部件的方法。有界上下文是进行成分划分的极好界限。根据定义，有界上下文将域中的区域彼此隔离。因此，按照这个想法，我们可以沿着相同的分界线隔离应用区域，将这些区域划分成单独的微服务。

有时候，这些微服务可能包含一些共享元素，在某些情况下需要一些通用的库来共享代码。在 DDD 中，有界上下文可以通过所谓的共享内核来共享域元素，这个共享内核一般是一组常见的域元素，可以被多个有界上下文使用。在 Akka 中，集群应用程序之间传递的消息协议通常可以在共享内核中找到。

共享内核中定义的消息协议代表有界上下文或微服务的 API，它定义了该服务的输入和输出。根据微服务之间使用的通信机制，它可能包含一系列直接发送到其他 actor 的简单 case 类，也可能代表首先需要被序列化为 JSON 或其他格式的 case 类，然后通过 HTTP 调用发送到 Akka HTTP 端点。

在调度域示例中，我们定义了三个有界上下文：用户管理上下文、调度上下文和技能上下文。可以将它们分为三个微服务，然后定义一个 HTTP 端点与这些服务进行交互，或者使用 Akka Cluster 或 Akka Remoting 直接向 actor 发送消息。

在将应用程序成功划分成微服务后，要开始思考如何扩展和部署应用程序。某些微服务

可能需要多个副本才能提供可扩展性，而其他微服务可能需要限制为单个副本。各个微服务的不同情况会影响我们设计和实现应用程序可用性的思路。

细粒度的微服务

虽然有界上下文是划分应用程序的完美起点，但有时并不够。有时我们需要将事物分解成比有界上下文更小的规模。Akka 的一个常见模式是将界面和域分成单独的微服务。在 REST API 中，这可能意味着 REST API 将作为单个微服务存在，而该 API 背后的实际逻辑可能是另一个。这是有利的，因为这样可以独立地扩展双方。

Akka 的好处是，我们可以做出这种类型的决策，而不影响应用程序的架构。位置透明性意味着我们可以决定是否将 REST API 放在同一个部署单元中，可以放在分开的不同部署单元中，而不影响应用程序的基本结构。

在下一节中，我们将探讨几种可用于分解应用程序的技术，这些技术旨在提供系统所需的可用性和可扩展性，至于哪个才是最适合的技术，取决于系统设计的具体细节。

集群感知路由器

当使用本地的单个 actor 系统进行系统构建时，路由器可以帮忙提供可扩展性。使用它们可以并行处理更多消息，从而在单个机器中扩展应用程序。处理分布式系统时，还可以将路由器作为一个提供可用性的工具。集群感知路由器与普通路由器相似，只不过它们的路由可能驻留在集群中的其他节点上，允许这些路由器在可扩展性之外提供可用性。

使用集群感知路由器可以启动集群中的多个节点。这些节点可以作为路由器路由的主机。在应用程序中，若向 actor 发送消息，可以通过集群感知路由器来实现。然后，路由器会确保其中一个节点将收到消息，但具体是哪个节点则取决于我们采用的路由策略。

如果一个节点出现故障，路由器可以简单地将消息路由到另外一个不同的活跃节点上。从客户端角度来看，不需要担心哪个节点收到消息，路由器会处理该细节。但是现在，我们获得了额外的可用性，节点可以离开集群而客户端甚至都不会有所感知。实际上，如果使用路由器池，当节点离开集群时，路由器池甚至可以在集群中的其他节点上创建新的路由，以补偿该节点的丢失。

需要注意的是，使用集群感知路由器时，其路由策略与本地路由器的路由策略不同。例如，使用最小邮箱的路由器在节点之间是没有传递任何关于邮箱大小的消息的，这意味着使用最小邮箱的路由器不知道集群中哪个节点有较小的邮箱。在这种情况下，路由器会对所有本地路由器使用正常的最小邮箱的优先逻辑，但远程路由的优先级就会比较低。

在哪里可以使用集群感知路由器呢？集群感知路由器是比集群单例更通用的构造，因此它的使用场景更加多样。工作者模式是使用它的一个很好的场景。在这种情况下，有一个 actor 可以将任务推送给多个工作者（worker）。例如，在示例域中，我们可以使用工作者模式进行调度，希望能够并行执行多个调度任务，特别是在涉及的工作量可能很大时。这种情况下，我们可以将消息推送到路由器中，然后将其分发给可用的工作者。这些工作者可以驻留在集群中的任何节点上。如果某个节点丢失，相关的工作者也会丢失，但是根据路由器配置，这些工作者会被重新分配到集群中的另一个节点上。

工作者模式的代码如下。

```
object Scheduler {
  case class ScheduleProject(project: Project)
  case class ProjectScheduled(project: Project)

  def props() = Props(new Scheduler)
}

class Scheduler extends Actor {
  import Scheduler._

  override def receive: Receive = {
    case ScheduleProject(project) =>
      // Do Work
      sender() ! ProjectScheduled(project)
  }
}
```

与集群单例一样，这里没有证据表明我们将使用 Akka 集群。我们可以稍后再做出决定。

创建工作人员的示例与创建本地 actor 的示例没有明显区别，代码如下。

```
val scheduler = system.actorOf(Scheduler.props(), "scheduler")
```

此示例创建了一个 actor，并将其命名为 scheduler。在本地系统中，可以像对待任何其他 actor 一样对待它，因为这只是一个标准的 actor。当引用这个 actor 时，神奇的事会发生在另一个集群节点上。

```
val scheduler = system.actorOf(
    ClusterRouterGroup(RoundRobinGroup(Nil), ClusterRouterGroupSettings(
      totalInstances = 100,
      routeesPaths = immutable.Seq("/user/scheduler"),
      allowLocalRoutees = false,
      useRole = None
```

```
    )).props(),
    "scheduler"
  )
```

此代码配置引用该 scheduler 的集群感知路由器组。我们可以看到它具有到 scheduler 的路径，与我们要路由的 actor 实例创建的路径相匹配。在这种情况下，最多允许 100 个实例被该路由器组使用，但是我们只创建了一个实例。如果有 100 个节点，那么每个节点都可以创建和管理 scheduler 的一个实例。我们还可以为路由路径指定不同的路径序列，以防止节点之间的模式发生变化。

此路由器通过排查集群中的节点来找到该 scheduler 的所有实例。然后，它会将消息路由到这些实例中，在这种情况下，它们使用轮询（round-robin）路由策略。然而，需要注意的是，与单例代理不同，路由器不会缓存消息。如果尝试在节点连接到集群并发现路由之前发送消息，则该消息将丢失。可以通过监听集群事件并在集群完全建立后开始发送消息来缓解此问题。当然，Akka 的交付保证仍然是“最多一次”的机制，所以如果真的需要可靠的交付，最好使用其他技术来实现。

分布式数据

在最终一致的分布式系统（如我们正在构建的系统）中，有时会遇到瞬态数据。这些数据不需要保存在数据库中，它们只存在于应用程序运行期间。如果应用程序被终止，这些数据可以被安全地清除，可能包括用户会话信息等。当用户登录时，我们需要存储有关该用户的信息：什么时候登录？使用什么安全令牌？最后一次活动是什么？这样的信息是有趣的，但持久化保存下来并不总是有价值的。用户注销后，该信息会变得无关紧要，因此需要进行清理。

同时，瞬态信息在所有节点上保持可用性对系统来说是非常重要的。如果用户遇到网络问题并断开连接，那么重新连接时则会连接到另一个不同的节点上。如果该用户信息在新的节点上不可用，系统将无法继续为该用户提供正常的服务。这种场景需要一种解决方法，该方法能够在集群中的多个节点上维护数据，同时不把该数据保存到数据库中。

有一种方法可以实现上述要求。如果可以使用以下特定标准的数据结构来表示数据，则可以以最终一致的方式可靠地复制数据。这种复制可以在内存中发生，不需要涉及数据库。这为我们提供了一种分布式的、最终一致的存储和检索数据的方法。

这些最终一致的数据类型称为无冲突复制数据类型或 CRDT。CRDT 是一个新兴的概念，首先出现在 Marc Shapiro 等人在 2011 年发表的论文中。目前，CRDT 还没有被广泛使用，但人们对它的兴趣正在增加，特别是在系统处理数据量非常大的数据流且要求高性能效

率的情况下。

Akka 实现了称为分布式数据的 CRDT。这是 Akka 的一个新模块，目前还存在一定的不稳定性，API 也经常会更改，因此开发人员还在讨论让它更好工作的方法及细节。

Akka 的 CRDT 思想的核心在于数据类型，包括计数器、集合、映射和注册器。为了被认为是 CRDT，这些数据类型必须包含一个无冲突的合并方法。此方法具有接收两种不同状态的数据（来自集群中的两个不同位置）并将其合并在一起以创建最终结果的能力。如果合并完成没有遇到冲突，那么就可以使用此数据结构来跨节点进行复制。

这是 CRDT 的工作原理。当一个节点接收到对数据进行更新的命令时，它将其当前状态广播给其他节点。其他节点接收到更新的状态后，将其与自己的状态进行合并，然后存储最终结果。

CRDT 通常通过与状态一起存储额外信息的方式来工作。添加操作通常是安全的，但删除操作就变得复杂了。例如，我们尝试删除集合中的一个元素，但是系统还没接收到将这个元素添加到集合的命令，所以并没有把它加入的集合中，这种情况下会发生什么？如何解决这个冲突？通常的处理方法是在这类元素的后续添加操作中做一些删除标记。与其直接删除这个还不存在的元素，不如把这个元素标记为已删除的状态，但是它还是会一直存在于系统的数据中。这意味着数据在不断增长，即使尝试删除信息，实际上系统还是在添加信息，只是这些信息被标记为删除状态而已。有些情况下，也可以结合一些优化方法来一起使用。

那么如何使用 Akka 分布式数据呢？来看一看如何在集群中的节点之间复制一些简单的会话信息。在以下例子中，假设我们只复制会话 ID（一个包装 UUID 的自定义类型），使用 ORSet 数据类型进行复制。ORSet 或 Observed Remove Set 是一种特殊类型的 Set，使用版本向量来跟踪元素的创建。然后，该版本向量作为合并函数的一部分来确定顺序。在 ORSet 中，如果添加和删除发生乱序或两者同时发生，可以使用版本向量来解决冲突。我们无法删除一个还未被添加的元素，如果添加和删除同时发生，则应该优先处理添加操作。代码如下。

```
case class SessionId(value: UUID = UUID.randomUUID())

object SessionManager {
  case class CreateSession(sessionId: SessionId)
  case class SessionCreated(sessionId: SessionId)

  case class TerminateSession(sessionId: SessionId)
  case class SessionTerminated(sessionId: SessionId)
```

```
  case object GetSessionIds
  case class SessionIds(ids: Set[SessionId])

  def props() = Props(new SessionManager)
}

class SessionManager extends Actor {
  import SessionManager._

  private val replicator = DistributedData(context.system).replicator
  private val sessionIdsKey = ORSetKey[SessionId]("SessionIds")
  private implicit val cluster = Cluster(context.system)

  override def receive: Receive = {
  case CreateSession(id) =>
    replicator ! Update(
      sessionIdsKey,
      ORSet.empty[SessionId],
      WriteLocal,
      request = Some(sender() -> SessionCreated(id))
    ){
      existingIds =>
        existingIds + id
    }

    case UpdateSuccess(
      `sessionIdsKey`,
      Some((originalSender: ActorRef, response: SessionCreated))
    ) => originalSender ! response

    case TerminateSession(id) =>
      replicator ! Update(
        sessionIdsKey,
        ORSet.empty[SessionId],
        WriteLocal,
        request = Some(sender() -> SessionTerminated(id))
      ) {
        existingIds =>
          existingIds - id
      }

    case UpdateSuccess(
```

```
        `sessionIdsKey`,
        Some((originalSender: ActorRef, response: SessionTerminated))
      ) => originalSender ! response

      case GetSessionIds =>
        replicator ! Get(sessionIdsKey, ReadLocal, request = Some(sender()))

      case result @ GetSuccess(`sessionIdsKey`, Some(originalSender: ActorRef)) =>
        originalSender ! SessionIds(result.get(sessionIdsKey).elements)
      }

  }
```

从以上代码中可以看到，为了复制数据，我们需要访问复制器。可以通过对 DataReplication 进行扩展获得特殊的 actor。可以通过向复制器发送更新消息来更新复制数据。数据更新后，我们将收到一条 UpdateSuccess 消息。可以通过向复制器发送 Get 消息来检索复制数据，复制器将使用 GetSuccess 消息进行响应。

还有其他可以发送给复制器的不同消息。订阅更新消息后，若系统的某些值发生变化，我们将收到订阅消息的通知，也可以使用 Delete 消息来删除记录。

复制器将负责在整个集群中复制状态。最终 actor 在集群中将保持一致。这意味着在运行 SessionManager 集群的所有节点上，请求 SessionId 列表都将获得活动会话列表。当然，因为这是最终一致的，所以节点在某些时刻可能会反馈略有不同的列表，可以通过指定不同的读 / 写一致性值来控制最终的一致性。

上一个示例中指定了本地一致性，表示复制器仅查看本地节点。也可以指定更严格的值，使复制器确保一定数量的节点在被认为有效之前必须与结果达成一致。ReadAll 和 WriteAll 表示所有节点必须同意，它会影响系统的可用性。如果节点发生故障，复制器将无法达到所需的一致性，并且请求将失败。还可以指定 Read Majority 或 WriteMajority，只要大多数节点同意该值即可，这为一致性与可用性提供了良好的平衡。

优雅降级

将应用程序分解为微服务的好处之一是可以实现优雅降级。在应用程序中，我们通过设置故障区域来实现正常的降级。通过在层次结构中创建 actor 可以使应用程序在面临一部分失败时不会导致整个系统崩溃，我们宁愿应用程序服务降级，也不愿它彻底崩溃。微服务器能够实现这样的要求，但它可能被会分散在多台 Java 虚拟机（JVM）或多台不同的机器上。

有一句老话说："如果不敢迈出第一步，那么跳伞也许根本不适合你"，我们希望应用程序即使遇到故障也能继续下去，而不要面临这句老话中暗示的情境。

在调度域的例子中，我们可以划分出几个不同的服务，包括调度引擎服务、项目管理服务、人事管理服务和技能服务。如果调度引擎服务失败，它不会影响添加或删除项目及人员的操作，只会影响我们安排相关人员到某个项目中。可以在单个单一应用程序中通过使用 actor 来创建故障区域，也可以使用微服务器来实现。

优雅降级表示，尽管应用程序的某些部分可能会失败，但应用程序整体上可以继续运行，即使面临故障也可以保持可用状态。这也意味着，为了易于维护，应用程序的非关键部分可能会被删除，而不会以不利的方式影响用户。

来看以下这个例子。系统中的调度引擎服务不一定需要快速被响应，创建新项目可能需要花费一段时间，几分钟甚至几小时，这是合理的。我们也没有预期一个项目在创建时就应该立刻被安排好。如果应用程序的调度部分需要进行维护（例如数据库升级），则可以将整个调度引擎服务关闭并执行维护。与此同时，系统仍然可以添加新的人员，还可以添加新的项目，在调度引擎服务维护完成之前，系统将不会调度新项目或分配给新人员。

可以使用第 7 章中讨论的熔断器模式来实现优雅降级。通常，检测外部系统的故障是一个耗时的操作。系统可能需要等待连接超时。如果对于每个后续请求都要这样处理的话，直到资源再次可用时，整个系统将承担许多额外的负担。通过熔断器，系统能够在第一次检查出问题的时候就快速拒绝任何其他后续请求，直到超过配置好的拒绝请求的时间。这有助于避免后续失败的请求消耗太长的时间，同时也减少了系统的负载，便于系统故障恢复。同时，这样也可以提高系统的可用性，因为系统快速通知了我们有关错误和不可用服务的信息，而不是提供耗时的超时警报。虽然系统的一部分不可用，但它仍然是有响应的，它只通知我们发生了系统错误。

部署

在使用上述技术构建应用程序并成功支持了可用性后，还需要以能够维护可用性的方式部署应用程序。如果部署过程要求关闭整个应用程序，那么便说明尚未达到预期的目标。

如果在部署过程中能有一些规划和预想，我们离服务 100% 的可用性便会更加接近。

在 Akka 中，每个可执行进程通常都包含一个单独的 actor 系统，该系统与集群中的其他进程相连，用来创建一个单独的分布式 actor 系统。

分阶段部署 / 滚动重启

导致节点在系统中不可用的最常见的原因并不是错误。相反，是因为一个正常的操作：升级操作。当应用程序或服务有新版本时，需要在某些特定时间将其部署到生产环境中。

自动化在这里至关重要，特别是在处理微服务时。一般不会只有单个应用程序的单个副本需要部署，往往会有多个微服务的多个副本，在大型系统中甚至有数百个。如果是手动操作，这将是一个乏味且容易出错的过程，但是借助自动化便可以使其更简单，更方便。

分布式系统最常用的方法是使用分阶段部署，也称为滚动重启。滚动重启的基本过程如下所示。

1. 选择集群中的一个节点。
2. 停止将新流量路由到该节点。
3. 允许该节点完成所有的现有请求，这被称为在节点上“排水”。
4. 在该节点上正常地终止原本计划升级的应用程序。在 Akka 集群中，这将涉及从集群中删除节点。
5. 将新的可执行文件复制到该节点。
6. 启动新的可执行文件，在 Akka 集群中，节点需要加入到集群中。
7. 验证节点是否可用，并且验证它是否为预期版本（监视状态页面在这里可以派上用场）。
8. 将流量路由到该节点。
9. 对其他节点重复以上步骤。

滚动重启要求有多个服务实例运行以保持可用性。当一个节点被选择进行部署操作时，其他节点需要保持可用性以便处理系统请求。

如果使用集群单例，则在升级期间无法维护可用性：单例模式不允许。但是，可以通过允许其他节点托管单例来将停机时间最小化。在这种情况下，关闭当前节点时，单例可以转移到某个其他主机上，从而缩短停机时间。

蓝 / 绿部署

滚动升级过程的替代方案被称为蓝 / 绿部署。对于此方法而言，系统拥有的节点数量必

须比实际需要的节点数量多。通常，正常操作所需的节点数量将比实际需要的节点数量多一倍，只有一半节点在给定的时间内为系统请求提供了服务。

在这种部署模式中，我们将50%的节点指定为“蓝色”，另外50%指定为“绿色”。假设绿色节点集群当前处于联机状态下并处理请求，那么便可以关闭所有的蓝色节点，升级它们，使其备份，然后检查它们的运行状态（记住，没有流量被路由到蓝色节点），最后交换绿色集群和蓝色集群，使蓝色节点处于活动状态，绿色节点处于服务不可用的闲置状态。接着，使用相同的过程升级绿色节点。

这个过程的一大优点是，如果新激活的设备集群在切换后出现问题，则可以在闲置集群被更新升级前重新切换回去（在本例，中上一个闲置集群是绿色节点集群）。因此，在新版本集群运行时，将闲置集群保留在旧版本中并观察一段时间是有好处的，这段时间内可以观察新版本服务是否存在问题。只有确信新版本是稳定的，才能升级闲置的集群。

崩溃恢复 / 运维监测

集群环境可用性的关键指标是能够识别出故障是什么时候发生的，然后对其进行适当的处理。更好的是可以在发生问题之前识别问题，并采取措施来阻止问题发生。要做到这一点，我们需要运维监测。可以使用各种监控工具，并且每个监控工具都在系统中起到不同的作用。

健康检查和应用状态页面

大多数监测工具都有一个共同点：依赖某种健康检查机制。执行健康检查的最常见机制是使用 HTTP 状态页面。

应用程序状态页面基本上是一个接受 `Get` 请求的 URL。它收到请求时会执行必要的内部检查，然后返回检查的结果，指示应用程序是否健康。返回结果可能是 HTML、JSON、XML 或任何方便当前业务需求格式的数据。

Akka HTTP 可以提供简单轻便的状态页面来帮助我们。事实上，即使应用程序不需要 HTTP 接口，包含 Akka HTTP 健康检查页面也是一件好事。完全使用 actor 通信的微服务器也可以从用 Akka HTTP 端点搭建的监控中受益，就和正常的维护一样。

那么健康检查可能是什么样的呢？这样的页面应该包含哪些信息呢？

最好包含一个指示服务外部依赖是否可用的信息：服务可以与其数据库进行通信吗？它可以与其他外部 API 通信吗？健康检查不需要很麻烦的操作，一个简单的 ping 类型操作就可以完成。根据服务的不同，外部依赖服务的失败可能不会导致应用程序的故障，它

还可以继续运行。我们需要根据具体情况判断失败的依赖关系是否会造成应用程序的故障，还是仅仅会发出警告或警报。

此页面中包含的另一个有用信息是应用程序版本号或提交哈希。这对于确定升级是否成功是非常有帮助的，也可用于跟踪错误。如果知道版本号，则可以排除不属于该版本号的更改，这可能会有助于确定问题的根源。

一个简单的监视页面的示例如下。

```
case class Symptom(description: String)

sealed trait Diagnosis {
  def name: String
}

case class Healthy(name: String) extends Diagnosis

case class Unhealthy(name: String, symptoms: Set[Symptom]) extends Diagnosis

case class HealthReport(versionNumber: String, healthChecks: Seq[Diagnosis])

trait HealthCheck {
  def checkHealth(): Diagnosis
}

class HealthCheckRouting(applicationVersion: String, checks: Seq[HealthCheck])
  extends HealthReportProtocol {
  val routes = {
    path("health") {
      get {
        complete {
          HealthReport(applicationVersion, checks.map(_.checkHealth()))
        }
      }
    }
  }
}
```

此路由类将提供健康监控页面。该页面将构建一个 `HealthReport`，其中包含版本信息以及应用程序中已实现的 `HealthCheck` 特性的各种服务状态。这些健康检查的实现细节会根据不同的服务有所不同，但基本思想是一样的：当进行健康检查时，如果有任何问题，它们将作为 `Unhealthy Diagnosis` 中的 `Symptom` 返回。如果服务是健康的，将返回 `Healthy Diagnosis`。`HealthReportProtocol` 接收结果报告，并将其转换为适当的

格式（例如 JSON、XML 等）。

度量

健康检查是一个很好的工具，用于在发生问题时给出提醒，同时也可以使用自动化工具进行监控，以便在出现问题时采取适当的措施（如通知团队）。然而，它们往往是被动响应的，不是主动响应的。自动化工具通知我们发生了问题，而不是警告我们将会发生问题。那么，如何在发生问题之前检测到问题，以便可以采取必要措施来防止这些问题呢？

一种方法是使用服务指标度量系统，通常是一个时间序列数据库。它可以是一个特殊用途的数据库，例如 InfluxDB 或 Graphite，也可以是一些通用的数据库，如 Cassandra 或 SQL，可以用它们创建基于时间的集合。在任何一种情况下，当有了可视化工具可以查看系统指标时，或许就可以做到提前检测系统问题了。

基于 Akka 或其他方式的系统都有一个很好的经验法则：将入口点周围的计时器包裹到系统中。这些计时器本质上用来记录操作开始和结束的时间，计算差异值，然后将其存储在时间序列数据库中。对于系统执行的每个操作而言，我们将知道执行该操作所耗费的时间长度以及该操作发生的时间点。

使用这些信息可以查看应用程序中的行为模式。例如，可以看到流量何时处于高峰。可以看到操作何时开始加快或减慢。有了这些信息，我们可以注意趋势。常握趋势的最好方法是把数据放在一个可视化的图表里。图表比在表格中显示信息更具可视化。看着这些图表，大家可能会注意到，应用程序每秒钟达到一定数量的操作时，其后不久就会崩溃。或者可能会观察到，当操作耗时太长时，通常也会发生系统崩溃。

当与 CPU、内存、应用程序启动、部署等其他信息组合时，这种类型的信息可以显示我们之前可能没有注意到的内容。它是运维监控的关键部分，特别是在大型分布式系统中可以使用它来将系统中的某些事件与故障关联起来。大家可能会注意到，当操作耗费太长时间时，内存使用量将上升。然后，可以将此阈值与应用程序故障相关联。这样可以帮助我们得出结论：系统需要更多的内存，或者我们在应用程序中使用内存的方式有缺陷（例如内存泄漏）。另外，如果观察到应用程序一直运行良好，直到最近的一次部署之后才出现问题，则可以排查在新版本中引入了哪些更改导致了问题。

然而，图表和指标度量只是第一步。它们告知我们有些环节出了问题，但是并不一定会告知问题细节。其中一部分详细信息来自于人对数据的解读。人类非常擅长模式识别。虽然可以让机器去监控如阈值这样的状态，但有时检测识别问题并不是机器擅长的。有时候，人们观察一个不超过任何阈值的图形，并且仍然会检测识别出某些问题。也许我

们没有使用太多的内存，但是使用内存的方式已经改变了。也许系统并没有经历太多的负载，但它就是和正常的情况相比有或多或少的波动。虽然这些观察结果可能是无意义或巧合的，但往往是即将发生问题的一个迹象。因此，有必要让人们时不时地观察这些图表，去发现这些偏离正常状态的异常。

日志

在发现图表中的内容发生异常之后，下一步就是确定原因。这也是日志表现得至关重要的地方。图表和健康状况检查可以提醒我们出现了问题，日志则可以诊断问题并分析原因。

与指标度量非常相似，好的做法是在每一次系统有输入时记录尽可能多的信息。如果对系统进行了 REST 调用，请记录该调用的详细信息。如果顶级 actor 收到消息，则记录该消息。当然，如果应用程序抛出异常，请确保在某些地方记录该异常。

当应用程序行为不正常时，这些日志将引导我们到指定地方排查问题。需要确保提供尽可能多的可用信息。如果忘记包含日志记录，那么当有需要时，就无法得到可用的相关信息了。记录过多的日志总好过日志不够用，但是请注意，记录太多的日志会给判别有用的相关信息造成困扰。

看门狗工具

当有了健康检查、指标度量和日志时，下一步是引入工具来获取信息并根据信息自动采取对应的行动。

首先需要某种通知机制。这些工具可以监控健康状况检测，甚至是日志记录和服务指标，并在发生问题时通过电子邮件、电话、聊天工具或其他机制来提醒我们。

虽然警报很重要，但它也可能令人困扰。我们都不想在周末的凌晨 3 点接到报警，尤其是报警通知系统发生故障需要修复。所以，如果一个工具不是仅简单地提醒故障，同时还可以采取一些纠正系统故障的措施，那将是更好的。

如果使用“让它崩溃”的思想构建系统，那么当应用程序崩溃时，一个合乎逻辑的方法就是重新启动它。有一些免费和商业可用的工具将提供此功能，如 Monit、Marathon、ConductR 等，我们可以监控应用程序，并且在发生故障的情况下自动重启它，甚至可能在集群中的不同节点上实现，而不需要面对一夜的电话报警。这种情况下无须发送通知，或者通知的性质可能会改变。系统可能只会发送一个电子邮件，而不是电话。

这也是一个要观看图表和日志的重要原因。根据配置的重启机制和故障问题的性质，系

统可能会持续几天不断重新启动，但仍然可以正常处理服务请求。这对系统来说不是一个很好的状态。如果没有注意到通知、图表和日志记录，我们可能永远也发现不了系统出现了问题。没有人会打电话投诉，因为从外部使用者的角度来看，系统正在正常工作。

运维监控并不是可以完成所有工作的单一的工具。有许多不同类型的监控，每个都提供不同的功能集和不同类型的信息。组合使用这些工具才能真正了解系统的情况和流程。健康状况检查和指标度量提醒我们出现了问题，日志则诊断具体问题，看门狗工具可以自动响应处理问题。每个工具都是有作用的，它们组合在一起能发挥最大的作用。

结论

本章介绍了基于 actor 的系统的不同要求：只记录日志是不够的，只有部署前的静态测试也是不够的。

通过适当的监控可以确保可用性，即使负载迅速变化（系统本身不断升级），系统的整体健康状态也可以立即显现。

在拥有所有这些好处的同时要保持高性能的话，还需要特别留意一些其他事情。这将是第 9 章的主题，也是构建基于 actor 的系统所需的最后一个关键步骤。

第9章

性能

优化 Akka 应用程序性能的方法和优化任何其他应用程序一样：首先隔离出应用程序里面比较耗时的部分，然后尝试对其优化，减少需要耗费的时间成本。

鉴于我们总是用有限的时间来进行优化，所以从最耗时的部分下手是最有意义的。当然，如果没有对系统进行性能测量，很难知道哪部分最耗费时间。因此，像所有应用程序一样，优化 Akka 的第一步通常是对系统进行性能测试，确认哪里耗时较长。

Java 虚拟机（JVM）中具有许多出色的测试性能和延迟的工具，可以将其中的任何一个应用于 Akka 应用程序内。例如，可以像测试任何非 Akka 应用程序一样通过调用 HTTP 接口使用 Gatling 工具测试 Akka 应用程序，并且可以将 ScalaMeter 应用于特定的代码段。

除了通常的服务请求数、延迟等统计数据，还有一些针对 Akka 的特有的统计数据也会起到帮助。

鉴于 Akka 应用程序中的慢消费者可能会成为严重的运行时问题，我们需要测量邮箱大小，以确保所有 actor（或 actor 组）的实际邮箱大小不会超出合理范围。

消息被给定的 actor 处理所需要的时间也非常有用。当与消息的频率信息结合使用时，它可以帮助我们缩小系统的瓶颈。例如，如果某个消息（如 `ComputeInvoice`）需要耗费相对较长的时间（通过 `ScalaMeter` 微量基准测量），但它并不是常见的消息，也不是对时间特别敏感的元素，那么它就不是一个需要优化的对象。一个更常见的消息是 `LookupInvoice`，它只需要相当短的处理时间，但是体量很大，可能是一个更需要优化的对象。

如我们所见，我们需要能够模拟应用程序的典型负载，甚至可以预测极端负载的样子，以便可以在现实条件下进行测量。

隔离瓶颈

有一个老故事，是关于一家工厂面临一个巨大的机器故障的问题。工厂经理打电话叫来一个专家，该专家直接取出机器的一部分，拿出一个月牙形扳手，然后转了四分之一圈拧紧了一个螺栓，机器立刻恢复了正常，专家离开了。几天后，工厂收到了专家发过来的5000美元的单据。工厂经理打电话给专家，抱怨说："五千美元？你在这里只待了五分钟而已！"专家冷静地回答说："我会给你发送一个新的单据和收费明细。"不久之后，另一个单据到达，内容是：拧紧螺栓，5美元；找到哪个螺栓需要拧紧，4,995美元。

Akka的优化和这个故事有点类似：通常只需要做一个微小的改变就能获得需要的结果，但是知道要在哪里做出改变是关键的。

优化性能的第一步是识别瓶颈。是什么资源耗尽才导致系统不能按照我们想要的方式执行？是内存吗？是线程还是内核？是I/O吗？

为了识别瓶颈，通常需要结合日志、监控和实时检查，VisualVM或YourKit等工具对此来说至关重要。

只有在消除了JVM调优中所有常见的可疑点（内存不足、对象泄漏、文件句柄耗尽），并且知道是项目中Akka部分包含的瓶颈，才可以考虑优化Akka本身。

请记住，JVM优化问题可能更加微妙：例如，垃圾收集器占用了太多的处理时间，或太频繁地进行"系统颠簸"，这些都不像普通故障那么明显，可以被看到。JVM调优超出了本书的范围，有许多其他的好资源可以用来学习。

优化Akka

在确定了一个潜在的可以优化的地方后，该选择进行哪些处理呢？可以处理的方面有三个，下面依次来看一看。

减少或隔离阻塞型操作

即使在基于actor的高并发应用程序中，仍然会存在许多阻塞型操作。例如，actor可能需要访问非异步的JDBC数据源。尽管最常见的阻塞操作是基于IO的，但还有一些其他阻塞操作可能占用当前线程，从而减少可用于其他进程的线程数，同时减缓整个系统的速度。下面讨论派发器时，我们将进一步阐述这一点。

缩短消息处理时间

由于 actor 主要是关于消息处理的，所以我们做的所有可以减少消息处理时间的事情对整个系统来说都是有好处的。如果可以将问题分解成能够同时解决的较小的问题，这通常是一个很好的策略，附加好处是可以将逻辑分成更小的、隔离的部分。actor 的创建开销很低，而且 actor（特别是在同一个 JVM 上）之间的消息通信也不需要太多开销，所以把一个问题从一个 actor 分散到一组 actor 一般是一个很好的策略，可以增加并行性。稍后详细讨论。

增加处理消息的 actor

如果情况允许，另一个策略是增加处理消息的 actor 的实例数。例如，在一组相同的 actor 之间使用路由器池来分发工作。这只有在问题被计算资源限制的情况下才起作用，即不受外部 I/O 约束，并且消息的每个实例是完全独立的（不依赖于要处理的 actor 中包含的状态）。这种技术的极限在于可用核心的数量。增加太多的 actor 无济于事，尤其是超过系统需要的数量时。然而，可以在一个集群中分配 actor 池，这样可以有效地获得比单个机器更多的核心。这是一个常见的 Akka 模式，如果正确应用，这种用法将成为 Akka 分布式系统中的大部分功能的基础。

与在一组相同节点之间简单地对请求进行负载均衡的技术不同，分布式 actor 可以应用多个节点的计算能力来处理单个请求，这是一个简单的节点均衡的解决方案所无法做到的。

派发器

派发器是优化 Akka 的关键组件。可以将派发器视为让 actor 系统运行的引擎。没有它们，系统不会运行。如果不细心使用它们，系统也无法正常运行。

派发器的工作是管理线程，将线程分配给 actor，并给 actor 机会处理它们的邮箱。它们如何工作取决于派发器的类型以及派发器设置的配置信息。那么有哪些类型的派发器可用呢？

标准派发器

标准派发器是最常用的。这是一个很好的通用派发器，它使用可自定义的线程池，这些线程将由派发器管理的所有 actor 共享。受 actor 的数量和线程数量的影响，actor 可能会被限制对线程的访问，因为那些线程可能正在被其他 actor 使用。

标准派发器的简单配置如下。

```
custom-dispatcher {
  type = Dispatcher
  executor = "fork-join-executor"
  fork-join-executor {
    parallelism-min = 4
    parallelism-factor = 3.0
    parallelism-max = 64
  }
  throughput = 5
}
```

在上面的示例中，吞吐量设置为 5，这意味着当前 actor 占用的线程在可用于其他 actor 前最多可以处理 5 个消息。这个系数可以根据具体需求设置为不同的值。

该示例使用了 fork-join-executor，然而，这不是唯一的选择。还可以使用 thread-pool-executor。根据使用的方式不同，参数也略有不同。可以如下使用 thread-pool-executor。

```
custom-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executor {
    core-pool-size-min = 4
    core-pool-size-max = 64
  }
  throughput = 5
}
```

每个 executor 都有一个最小和最大线程值，以确保可用线程的数量保持在合理范围内。每个 executor 都有一个系数，这个系数是一个乘法系数。如果要计算可用线程数，可以用这个乘法系数乘以机器上的核心数，然后将其限制在最小和最大值之间（如果计算结果小于最小值就取最小值，大于最大值就取最大值），这样得到的值就是可用的线程数了。这意味着，如果将应用程序移动到具有更多内核的机器上，则会有更多的可用线程，前提是最小 / 最大的线程数值允许。请注意，使用 fork-join-executor 的时候，parallelism-max 不是最大线程数，相反，它是活跃线程的最大数值。这是一个重要的区别，因为 fork-join-executor 可以在阻塞时创建新线程，导致线程数激增。

使用 fork-join-executor 还是 thread-pool-executor 主要取决于该派发器中的操作是否会被阻塞。fork-join-executor 可以提供最大数量的活跃线程，而 thread-pool-executor 可以提供固定数量的线程。如果线程被阻塞，fork-join-executor 将会创

建更多的线程，而 thread-pool-executor 则不会。对于阻塞型操作，通常选择使用 thread-pool-executor 会更好，因为它会防止线程数量激增。fork-join-executor 更适合“响应式”操作。

Throughput 是一个有意思的选项。Throughput 的值决定在派发器尝试释放当前 actor 对线程的占用前允许该 actor 处理的消息个数。它决定了 actor 在共享线程中的公平性。像“1”这样的低吞吐量值表示 actor 之间共享线程可以保证尽可能公平。值为“1”表示每个 actor 将只能处理一个消息，然后就将让步线程给另一个 actor 使用。另一方面，吞吐量值为 100 就会不公平，每个 actor 在线程让步前将处理 100 条消息。什么时候应该使用高吞吐量值，什么时候使用低吞吐量值呢？我们需要在系统中尝试各个数值才能找到最佳的值，除此之外其实还有一些其他事项需要考虑。

每一次线程让步都会执行一次上下文的切换。如果经常发生上下文切换，对于系统来说是一个比较昂贵的开销。根据应用程序和消息大小的不同，太多的上下文切换可能会对应用程序的性能产生重大影响。如果一直有大量消息流快速流经派发器，将会发生频繁的上下文切换，进而对系统产生非常大的负面影响。这种情况下，我们可能会想把吞吐量值调高。因为消息很快，不会有 actor 被闲置很长时间，所以尽管 Throughput 被设置得比较高，它们的等待时间还是会比较短。并且通过增加 Throughput 的值，可以允许系统最小化上下文切换带来的影响，这有助于提高系统的性能。

但是，如果消息产生的速度并不快，并且需要很长时间才能处理一条消息，那么高 Throughput 值可能会严重影响系统性能。这种情况下，我们会发现大部分 actor 都是空闲的，因为它们正在等那些缓慢的消息被推送过来。这时，上下文的切换已经不是影响系统性能的主要因素了，因为那些缓慢的消息需要等待更长的时间。这里为了公平性需要调整派发器。这时候低吞吐量值可能是我们更想要的，它允许长期运行的 actor 处理单个消息后立刻将线程让步给另外一个等待的 actor。

在代码中使用派发器就是简单地在 actor 的 Props 上调用 withDispatcher 方法，代码如下所示。

```
val actor = system.actorOf(Props(new MyActor).withDispatcher("custom-dispatcher"))
```

固定派发器

在某些特殊的情况中，共享线程可能会对 actor 系统造成不利影响。在这种情况下，可以选择使用固定（pinned）派发器。一个固定派发器不再为 actor 使用线程池，而是直接为每个 actor 分配一个线程。这样消除了调整吞吐量和线程数量的担忧，但同时会产生一些其他的代价。

线程并不是免费的。运行这些线程的硬件数量往往是有限的，它们需要共享硬件。这种硬件共享有它们自己的上下文切换形式。大量的线程意味着更多的上下文切换，因为这些线程需要竞争相同的资源，而且可以创建的线程数量也是有限制的。即便为每个 actor 分配一个线程的概念听起来很吸引人，但实际上它的作用比我们所期待的更为有限。通常情况下，最好的选择是使用标准派发器，而不是固定派发器。

使用固定派发器的另一个考虑因素是，每个 actor 只分配一个单线程。如果 actor 没有混合使用并发技术，这不是问题，但是 actor 经常会在内部使用 future。这样做时，通常使用 actor 的派发器将作为执行上下文，在这种情况下，actor 只被分配了一个线程。这意味着使用 actor 的派发器将只允许访问单个线程。根据实际情况，这可能会导致系统性能下降，因为 future 现在必须以单线程方式运行，因为系统的多个并发部分在竞争同一个线程资源，因此在某些情况下甚至会导致死锁。

那么什么情况下适合使用固定派发器呢？在有一个 actor 或少量 actor 需要尽可能快地运行时，固定派发器会很有用。例如，如果系统有一组被认为是“高优先级”的操作，那么我们可能不希望这些操作去共享资源。可能会希望避免使用线程池，因为高优先级操作将因为共享线程而彼此耦合（也可能是与其他 actor），它们会因为线程资源发生竞争冲突，这会导致它们被阻塞而变慢。在这种情况下，使用固定派发器可能是一个很好的解决方案。固定派发器意味着 actor 不需要彼此等待。

平衡派发器

平衡派发器会将忙碌的 actor 的工作重新分配给闲置的 actor。它通过使用共享邮箱来执行这样的重分配操作。所有消息都会进入同一个邮箱。派发器可以将消息从一个 actor 传递到另一个 actor，允许它们共享负载。

然而，适合平衡派发器的使用场景比较有限，很多场景下它都不是正确的选择。由于派发器的共享性质，不能将其用作通用邮箱。使用派发器的所有 actor 必须能够处理邮箱中的所有消息，这通常会限制其只能用于相同类型的 actor。

平衡派发器与标准派发器的工作方式大致相同。它使用一个线程池，所以可以使用与标准派发器相同的方式对其进行优化。

平衡派发器对于具有多个相同类型的 actor 并且希望在它们之间共享工作负载的情况可能是有用的，但是使用其他方法通常会是更好的选择。

calling-thread 派发器

calling-thread 派发器主要用于测试。这种情况下不会分配线程池。所有操作都在发送原

始消息的同一个线程上执行。它消除了造成测试困难的大量并发问题，这在测试中非常有用。当一切都在同一个线程上运行时，并发不再是问题。

当然，这种系统并发被消除的情况是人为的。在实际的系统中不会以单线程方式运行，所以即使这种模拟对测试有用，但它也可能会产生一个关于系统行为的错误反馈。更糟糕的是，它也可能导致 actor 发生死锁，因为它们突然发现自己需要异步等待，但却无法这样做。通常情况下，即使在测试中，使用适当的基于线程池的派发器也会更好。

此外，Akka TestKit 提供了许多测试工具套件，因此没必要使用 calling-thread 派发器。

何时使用单独的派发器

当创建一个 actor 而不显式为它分配派发器时，该 actor 会被创建为默认派发器的一部分。因此，它将与所有其他 actor 共享一个线程池。对于大多数 actor 来说，这是完全可以接受的。但是，某些情况下，它们也需要单独的派发器。

我们希望尽量避免在 actor 内执行阻塞型或长时间运行的操作。这些阻塞操作可能会对线程池中线程的可用性产生重大影响，并且会影响应用程序的性能。但是我们不可能一直都避免这类操作。有时候无论怎么努力尝试避免，还是必须涉及阻塞型操作。这种情况下需要使用单独的派发器。

下面来分析一下若不使用单独的派发器会发生什么。比较普遍的是在建立系统时提供特殊的“监控”或“健康检查”操作。这些操作相当简单，只是为了验证应用程序正在运行并且具有响应。它可以像 ping / pong 测试一样简单。或者它也可能很复杂，包括检查数据库连接和测试各种请求。目前，我们只考虑简单的 ping / pong 测试。这个操作非常快，不需要与数据库进行通信，也不需要进行任何计算，代码如下。

```
object Monitoring {
  case object Ping
  case object Pong
}

class Monitoring extends Actor {
  import Monitoring._

  override def receive: Receive = {
    case Ping =>
      sender() ! Pong
  }
}
```

如果在同一个 API 中有另一个更复杂的操作呢？此操作需要从数据库读取数据，进行数

据转换，最终返回。进一步假设我们使用的数据库驱动是一个阻塞型驱动程序，代码如下。

```
class DataReader extends Actor {
  import DataReader._

  override def receive: Actor.Receive = {
    case ReadData =>
      sender() ! Data(readFromDatabase())
  }

  private def readFromDatabase() = {
    // This reads from a database in a blocking fashion.
    // It then transforms the result into the required format.
    ...
  }
}
```

如果这两个操作在同一个派发器中执行，它们如何相互影响？如果同时有多个长时间运行的操作请求进入，这些请求可能会阻塞派发器中的所有可用线程，这些线程在操作完成之前将始终不可用。这意味着当监视服务已经启动 ping / pong 请求去验证 API 是否可用时，它将被延迟或失败。没有可用的线程执行 ping / pong 请求，所以服务必须等待。

在这个非常简单的例子中，这两个操作之间存在耦合。即使这些操作执行差异很大的任务，并且在许多方面是完全不相关的，它们还是会因为共享的资源（在这种情况下是线程池）而存在耦合。由于这种耦合，其中一个操作中的较差性能可能会导致另一个操作的性能也较差，这是不可取的。

要解决这个问题，可以引入一个单独的派发器。将执行每个任务的 actor 放在一个单独的派发器上可以解开此耦合。现在，每个操作都有了自己的线程池，因此当长时间运行的操作阻止其所有可用线程时，仍然会有一个单独的派发器可以提供执行 ping / pong 请求的额外线程。即使系统的其他部分已被阻止，监控也可以保持运行，代码如下所示。

```
data-reader-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executor {
    core-pool-size-min = 4
    core-pool-size-max = 16
  }
  throughput = 1
}

// For monitoring, because it is a very fast operation,
```

```
// it is sufficient to leave it in the default dispatcher.
val monitoring = system.actorOf(Monitoring.props())

// For our data reader, we have a separate dispatcher created to manage
// operations for reading data.
val reader = system.actorOf(
  DataReader.props().withDispatcher("data-reader-dispatcher"))
```

有很多情况会用到单独的派发器。单独的派发器可以独立调整 actor 组。如果一组 actor 需要设置高吞吐量，而其他组在低吞吐量设置下可以更好地工作，将它们分成不同的派发器可以进行独立调整。或者，如果有一组 actor 将会收到大量消息，另一组收到较少量的消息，那么我们可能希望将它们分开。优先级呢？如果某些消息的优先级高，则可能需要在派发器中隔离它们，使它们不与其他 actor 竞争线程资源。

关键是我们不想在任何地方使用同一个派发器。人们一开始使用 Akka 时最容易犯的错误便是使所有的 actor 都使用默认的派发器。这会导致系统的响应变得很慢，甚至失去响应，尽管事实上它此时并没有消耗大量的硬件资源，但最终可能会发现 CPU 基本上处于空闲状态，系统仍然没有响应。创建单独的派发器并针对特定用例进行调优是解决此问题的第一步。

提高并行性

提高并行性是优化 Akka 的另一种方法。通常，在简单增加处理请求的 actor 数量之前先提高并行性其实是很有帮助的，但提高并行性不一定意味着增加 actor 的数量。

在任何给定的算法中，如何提高并行性是一个大问题，可能需要一整本书来讲解，但一些基本原则是都通用的。通过减少耦合和资源共享来减少竞争是一个关键点，将问题分解为不依赖于其他问题的多个部分是另一个关键点。

即使我们能够重构算法以允许更大的并行性，如前所述，实际可以实现多少并行性仍然会受到单个节点上的可用核心数量的限制——这就是 Actor 模型超越以往限制的地方。

单个请求或算法可以将其可并行化的元素进行分布式部署，即可由多个节点处理。然后，每个节点都可以使用该节点上可用的所有核心，从而使总体并行性高于单个节点的并行性。

最简单的应用通常是那种受计算资源约束的算法，换句话说，它不能在单个节点上实现更快地处理，因为它已经耗尽了 CPU 资源。通过将该算法分解为不依赖于彼此的独立部分，然后分布式地部署它们，可以在更大的“机器”上更有效地计算问题，这要比任何单个节点都快速有效。

不是每个问题都受计算资源约束，许多是受I/O约束的，比如对某种持久化存储的数据访问。在这种情况下，可以应用分布式领域驱动设计（DDDD）模式，允许将数据与actor的实例一起包装并加载到内存中。如同受计算资源限制的问题通过应用该模式可以获取比任何一台机器提供的CPU资源都要多的资源一样，分布式域问题也可以通过应用它获得比任何一台机器提供的内存资源都要多的内存资源，并将一个I/O约束的问题转变为一个类似于计算约束的问题。

在任何情况下，通常都有持久化的需求——比如事件日志是由系统中的actor发出并处理的。如果actor要访问不同的节点，那么我们将再次受I/O约束，有时这个约束来自于网络而不是磁盘读写。

结论

在本章中，我们介绍了如何仔细地考虑和调整性能，避免了基于actor的系统在高度可扩展性和可用性方面的瓶颈。

大家现在已经了解了使用Akka和Actor模型构建应用程序的所有重要的环节，从最细粒度到最高级别的所有方面。

后记

我们将并发和分发的思想作为现代软件开发的基础，开始了整本书籍的写作。经典的计算模型为我们创造了一种不同于现实世界的，在分布式和并发环境中特别有价值的幻象。

尽管如此，我们已经看到，Actor 模型建立在不同范式的基础上，更符合实际场景，特别适合设计和构建组件的位置和行为彼此独立的系统。

actor 是一个非常强大的抽象模式，但它们与普通命令式甚至函数式应用程序开发方法完全不同。如果应用正确，它们可以提供轻松扩展应用程序的手段，无论是处理负载的能力还是处理复杂多级计算的能力都会有所提升。但如果应用不正确，应用程序会变得复杂而且难以理解，因此难以维护。

不同之处在于设计模式。

好的 actor 模式和好的响应式微服务架构之间有许多相似之处。响应式微服务架构被应用于“大型”的项目中，作为高层次抽象来构建系统；actor 模式适用于“小型”的项目，在单个独立服务内或在服务之间使用。

然而，如果用得好，两者都可以有隔离且独立的自主性、弹性、私有状态保护机制以及与系统其他部分的异步消息进行通信的属性。因此，由 actor 构成的服务可以相当自然地组成响应式微服务系统。

与微服务一样，为不同的元素选择合适的通信方法是关键，在保持隔离的同时找到正确的整合系统方法。

然而，像响应式或微服务系统一样，如果没有遵循具体的指导原则，基于 actor 的系统的附加成本可能会超过其产生的收益。在整本书中，我们试图指出某些模式可能具有更多的劣势而不是优势，希望各位能够节省亲自去经历这些劣势的成本。

这本书描述了最重要的 actor 模式，特别是在 Akka 库中使用它们的方法。我们相信，如果大家遵循并扩展这些模式，一定能创建具有可伸缩性、弹性、灵活性以及易于维护性的应用程序和系统。

参考文献

"Alan Kays Definition Of Object Oriented." Last modified April 18, 2014. *http://c2.com/cgi/wiki?AlanKaysDefinitionOfObjectOriented.*

"Alan Kay On Messaging." Last modified November 12, 2014. *http://c2.com/cgi/wiki?AlanKayOnMessaging.*

Allen, Jamie. *Effective Akka*. Sebastopol: O'Reilly Media, 2013.

Avram, Abel, and Floyd Marinescu. *Domain-Driven Design Quickly*. Lulu.com, 2007.

Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2003.

Hewitt, Carl, Peter Bishop, and Richard Steiger. "A universal modular ACTOR formalism for artificial intelligence." *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (1973): 235-245.

Hewitt, Carl, Erik Meijer, and Clemens Szyperski. "Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know...)." *https://www.youtube.com/watch?v=7erJ1DV_Tlo.*

Malawski, Konrad. "Lambda Days 2015 - Konrad Malawski - Need for Async(..)." *https://www.youtube.com/watch?v=Q_0z2v3QJg4.*

关于作者

Michael Nash 是 Lightbend 的总监。过去 30 多年间，他为各种规模的公司及客户设计、开发、发布了很多软件项目。作为项目管理和架构实践的软件工艺倡导者，他是“安全堆栈”最早的实践者之一，在 Scala、Akka 和 Spray 方面有 5 年以上的工作经验。他在大部分业余时间中致力于“安全响应式平台”的相关研究，在会议中发表主题演讲，以及在软件相关领域撰写书籍。

Wade Waldron 是 Lightbend 的高级顾问。他现在针对 Lightbend 响应式平台提供培训和咨询服务。在 Lightbend 之外，Wade 在过去 9 年中一直在设计功能强大的软件和游戏，重点是设计测试驱动、领域驱动、面向服务的架构、事件驱动架构以及敏捷开发。

封面介绍

封面上的动物是一种常见的水鸭，或称为欧亚水鸭（绿翅鸭），是一种在欧洲和亚洲广泛分布的鸭子。颜色 teal 就是以这类鸭子命名的。在繁殖季节，雄性欧亚水鸭的眼睛周围会长出生动的蓝绿色羽毛。这类鸭子是“钻水鸭”的一个分支，主要在地面或浅水中觅食。

欧亚水鸭分雌雄两性：雌性的身体主要是棕色的，伴有黑色的斑块，尾巴是白色的。雄性则是灰色的身体，上有较细的黑色条纹，棕色和白色的翅膀，栗色的头部，眼睛周围有蓝绿色羽毛。在繁殖季节之外，雄性的羽毛不那么生动（被称为“非婚羽”）。欧亚水鸭是最小的鸭子之一，平均体长 8 ～ 12 英寸，重约 12 盎司。

它们生活在水塘、小湖泊和湿地的淡水栖息地，在被茂密的植被深深隐藏的地面上筑巢。在繁殖季节（大约 3—5 月），它们吃小型甲壳类动物，如昆虫、幼虫、蠕虫和鱼苗等。而在一年中的其他时间，它们吃水生植被和其他植物。它们每个冬季迁往地中海、南亚和非洲，繁殖季节返回北欧和亚洲的温带地区进行交配。

O'Reilly 封面的许多动物都是濒临灭绝的，它们对世界来说很重要。要了解更多关于如何帮助濒危动物的信息，请访问 *animals.oreilly.com*。

封面图片来自伍德所写的 *IUustrated Natural History*。

O'REILLY®

Akka应用模式：分布式应用程序设计实践指南

涉及大数据处理时，我们不能再忽略并发性，也不能尝试事后再将其添加到系统中。幸运的是，与并发性相关的解决方案并不是一种新的模式，而是一种已有的开发模式。通过本书的学习，Java和Scala开发人员将掌握如何使用开源的Akka工具包进行并发性和分布式应用程序开发，大家将学到如何将Actor模型及其相关模式置于实际应用之中。

在这本书中，我们将处理类似劳动力分配的问题：如何在优化一组人员可用时间和技能的同时，安排他们执行其他的项目。这个例子将帮助我们了解Akka如何使用actor、streams和其他工具将应用程序拼接在一起。

- 通过域驱动设计对软件建模从而反映现实世界
- 了解实现单个actor的原则和方法
- 通过组合多个actor的模式发挥Akka的真正实力
- 了解分布式系统中的一致性权衡问题
- 几种Akka故障隔离和故障处理方法
- 探索构建支持可用性和可扩展性系统的方法
- 使用JVM工具和派发器优化Akka应用程序的性能

“这本书为那些准备使用Akka以及了解其背后响应式原理的JVM开发人员提供了全面的指南，是学习构建响应式系统基础知识以及探索Akka优越性能的一个很好的资源。”

——**Konrad Malawski**
Lightbend高级工程师

Michael Nash是Lightbend的总监。30多年来，他为各种行业和规模的客户设计、开发、发布了很多软件项目。

Wade Waldron是Lightbend的高级顾问。10多年来，他带领团队开发了一系列软件项目，从游戏到穿戴式设备，再到广告。目前，他正在针对Lightbend响应式平台提供培训和咨询服务。

PROGRAMMING / ENGINEERING

图书分类：程序设计

策划编辑：孙奇俏
责任编辑：张春雨

Broadview®
www.broadview.com.cn

O'Reilly Media, Inc.授权电子工业出版社出版

ISBN 978-7-121-32529-8

定价：65.00元